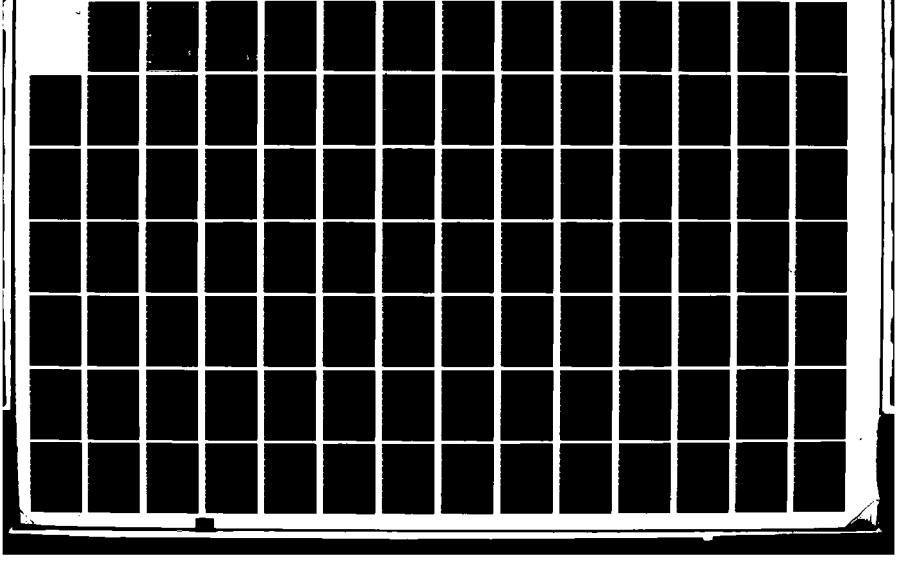
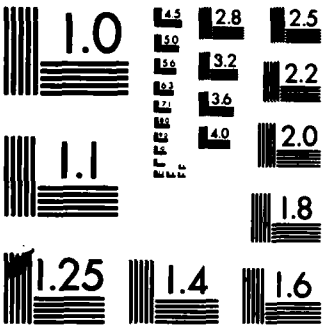
AD-A138 022 SIRE: AN AUTOMATED SOFTWARE DEVELOPMENT ENVIRONMENT(U) 1/2
AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL
OF ENGINEERING D W NETTLES DEC 83 AFIT/GCS/MA/83D-5
UNCLASSIFIED F/G 9/2 NL





SIRE: AN AUTOMATED SOFTWARE DEVELOPMENT ENVIRONMENT

THESIS

AFIT/GCS/MA/83D-5 David W. Nettles

1st Lt USAF

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/MA/83D-5

SIRE: AN AUTOMATED SOFTWARE DEVELOPMENT ENVIRONMENT

THESIS

AFIT/GCS/MA/83D-5 David W. Nettles

1st Lt USAF

Approved for public release; distribution unlimited

AFIT/GCS/MA/83D-5

SIRE: AN AUTOMATED SOFTWARE DEVELOPMENT ENVIRONMENT

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

by
David W. Nettles, B.S.
1st Lt USAF
Graduate Computer Science
December 1983

Preface

To the memory of

Dira Anne Nettles.

The sun doesn't always shine.

Table of Contents

List of Figures

Abstract

The objective of this thesis is to perform the preliminary design and partial development of an automated software development environment (ASDE). This environment, called Sire, is intended to support the design and production of software using automated and interactive tools. Sire is to be a system that aids the software designers and programmers through the use of an integrated and flexible set of tools that are intended to reduce the amount of work that is done by humans. This reduced workload will free the system designers/implementors for more productive work.

As part of this investigation, a partial implementation of Sire is accomplished. This implementation allows the user to input a system design in a specification language. Sire will then produce a correct source program shell for the user to use for the detailed implementation stage.

Chapter 1

Introduction

1.1 Thesis Objective

The objective of this thesis was to perform the preliminary design and partial development of an automated software development environment (ASDE). This environment, called Sire, was intended to support the design and production of software using automated and interactive tools. Sire was conceived to be a system that aids the software designers and programmers through the use of an integrated and flexible set of tools. Sire was designed and implemented with all facets of software production, use, and maintenance in mind. After the preliminary design was completed, a portion of of the design was chosen for implementation. The implemented portion of Sire provides for the development of the preliminary design and the automatic production of a preliminary programming language source program.

1.2 Background

1.2.1 The Software Crisis

In recent years, the term "software crisis" has often been used to characterize the state of the art in software systems development. The roots of the software crisis, and the key to understanding the problem, lay in the late 1950's and early 1960's. In those days, computer software was a new concept and people did not thoroughly understand it. Therefore, they did not recognize the need for the proper engineering and construction of software. Some symptoms of this immature industry were unresponsive products, slippage of production schedules, and difficulty in operations and maintenance of software (Infotech, 1977: 8). Many of these symptoms were directly attributable to the methods used to develop software with each adding unnecessarily to the costs of the final software product.

Although the cost of software development was high, initially there was little or no motivation to do anything about the cost. The actual computer hardware consumed the majority of the computer system budget as can be seen in Figure 1. Because of this, no one gave much thought to the relatively small budget share allocated to software

development. Furthermore, the hardware designers believed that the need for, and cost of, software would be greatly reduced by the development of more advanced hardware, and therefore more concentration was given to hardware research and development. It was this attitude that retarded the growth of the software industry as a science and kept it from maturing at a rate equal to that of the hardware industry.

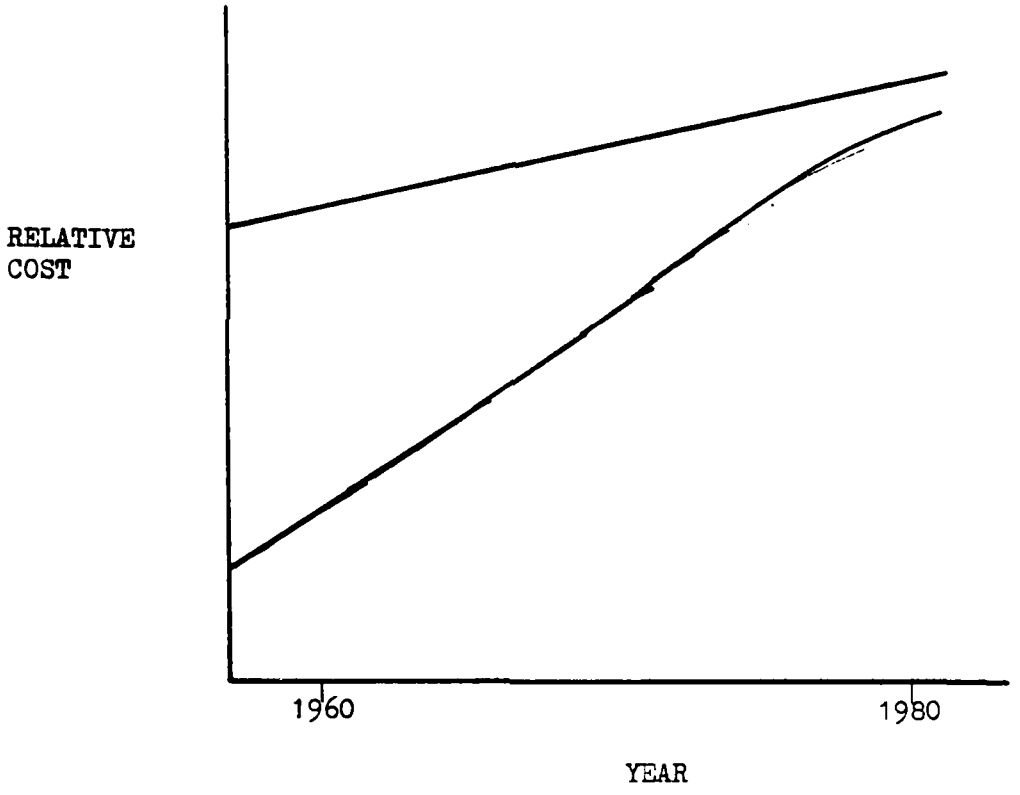


Figure 1. Proportional Cost of Hardware and Software

It soon became clear, however, that this course of events was leading towards disaster. As larger and more complex software systems were attempted, more and more software

project failures were noted. Even those systems that worked were often unreliable, poorly documented, inefficient, unable to fulfill the users' needs, and in need of costly rework (Wasserman, 1981: 16). The wonderful hardware that had been developed was not being efficiently used due to this failure to provide quality software. Fueled by these failures, the cost of software development started to rise dramatically. It was predicted that future software would soon become the predominant factor in the cost of computer systems, accounting for as much as 60% to 90% of computing systems costs as illustrated in Figure 1 (Infotech 1977: 7). With the common development failures and the predicted expense for software, it became increasingly clear that something needed to be done to assure better quality software at a lower cost.

1.2.2 Software Engineering

"Software design technology is a system - not a secret" (Peters, 1981: 3). While this may be true, it seems that the fact that there was a system was a secret until around 1968. It was at this time that the term "software engineering" was first used. This term was chosen to emphasize the idea that the proper design and construction of software should be viewed not as some mystical art, but rather as an engineering discipline (Wasserman, 1981: 16). Development of the discipline of software engineering was started in the early 1970's and continued mainly as academic exercises that had

little or no affect on software development until the mid to late 70's. As a result of these research efforts, there were many methods and techniques developed that were intended to aid the production of high-quality software systems. Today, the term software engineering is used to describe this collection of practices, techniques, and methods.

Software engineering literally encompasses all activities associated with producing software (Peters, 1981: 6). These activities are generally associated with some model of the software development process such as the software life-cycle model depicted in Figure 2. This "waterfall model" of the software development cycle is denoted by the neat, concise, and logical ordering of a series of steps that occur in order to complete and deliver a software product (Peters, 1981: 12). Although depicted as a step by step operation, each of these steps sometimes blur into each other during software development. They may also take place in parallel. Therefore, a more general model might be to break the development cycle into three phases as follows:

1) Analysis and Design
2) Implementation
3) Use and Maintenance

These phases encompass the waterfall model with the advantage that it is easier to discuss generalaties with this simple model.

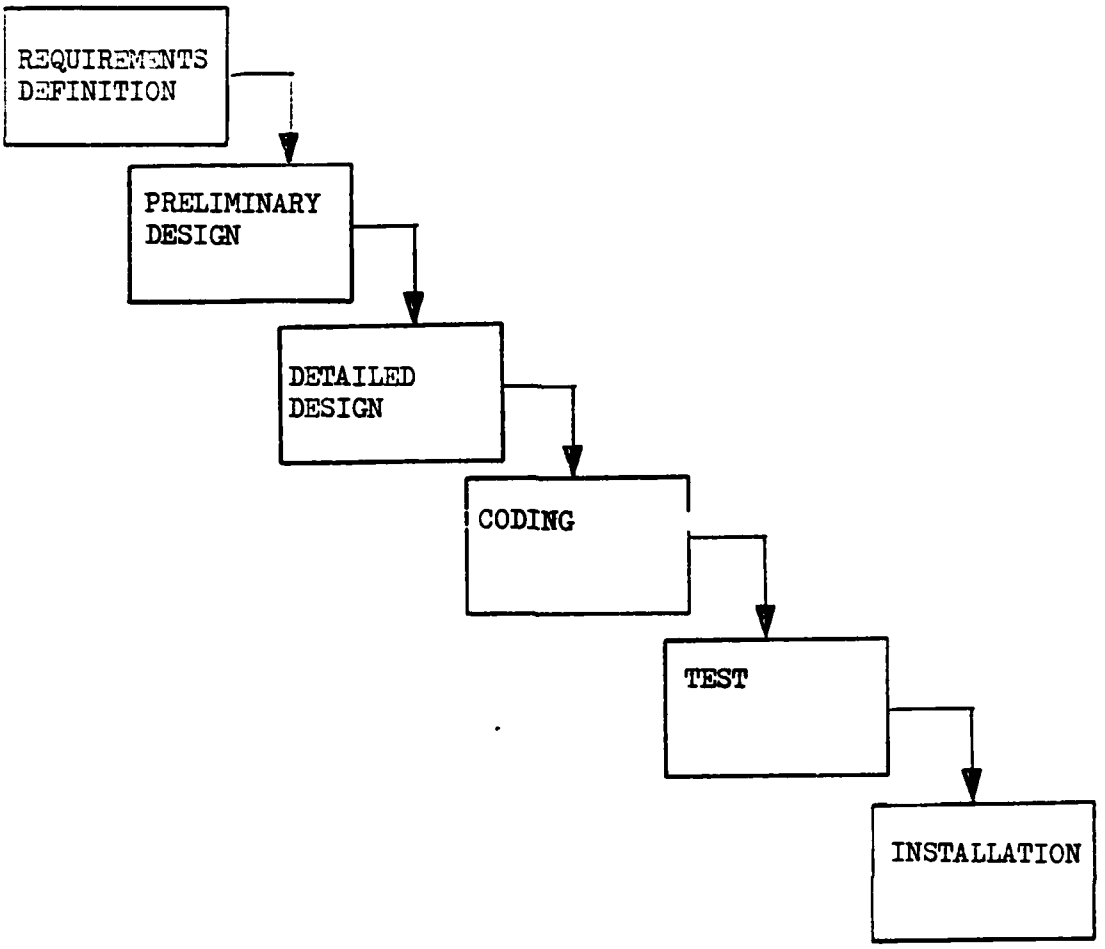


Figure 2. Software Life-cycle Model

Nearly all software engineering methodologies and techniques were developed with full support of the software life-cycle in mind. However, the key word in the last sentence is support." Some methods provide a greater degree of support for one phase and less for others. Commonly, software engineering techniques concentrate on the Design and Analysis phase with the belief that proper design can eliminate the majority of the errors that occur in software systems. Studies have shown that 64% of the errors that occur in projects are introduced in the Analysis and Design stage,

before the Implementation stage begins. The earlier these errors are detected, the less it will cost to correct them, as is shown in Figure 3. These techniques usually do little for the Implementation phase and provide even less support for the Use and Maintenance phase. The reason for this is that more support provided for each phase means more work for the user and more work means greater cost. Therefore, current software engineering methods properly put emphasis on the Analysis and Design phase in order to realize the greatest cost benefit.

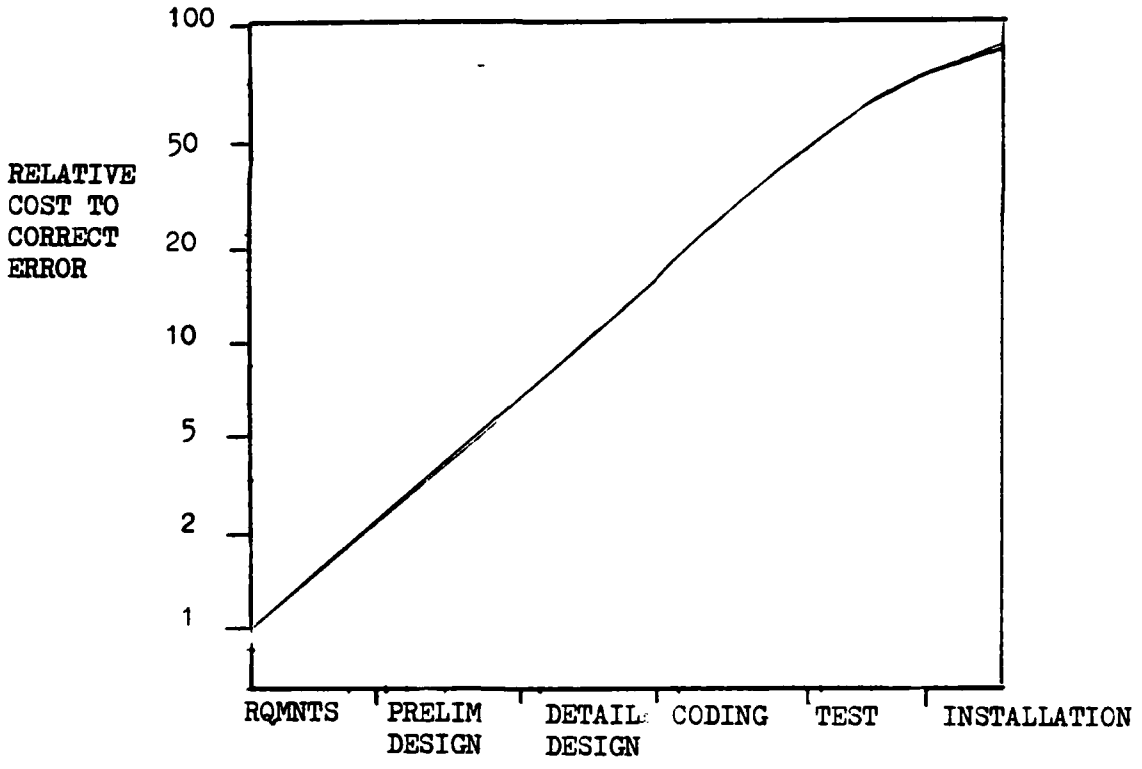


Figure 3. Error Detection Costs

1.2.3 Automated Environments

As has been noted, software engineering methods and techniques were developed to ease the software development process by the application of proper engineering methods. The problem with these techniques and methods is that, for proper use, they generally require the imposition of a discipline on the users. The discipline is necessary in order to make the users of the method follow all the steps outlined so that the end product will be as specified by the method. This discipline must be imposed by management and is usually hard to control because it generally implies a great amount of bookkeeping, illustrating, and consistency checking, and all these are usually an unpopular added burden to the people responsible for the tasks. Given this, and the realization that the methods and techniques of software engineering do not completely support the life-cycle, a major new thrust of software engineering research is in the field of Automated Software Development Environments (ASDE).

By using the full potential of the computer to automate one or more of the methods or techniques of software engineering the user can be relieved of much of the tedious tasks associated with the methodologies and the use of a methodology benefits the support of the life-cycle. Also, more support can be given to all phases of the life-cycle

since most of the work done in each phase is done by the computer. A software development environment is a collection and integration of software development tools that are intended to adequately support the entire software development life-cycle. The classic environment consists of tools such as text editors, debuggers, compilers, and linker/loaders. The problem with these tools is that although they are adequate for their intended purpose, they do not utilize the full potential of the computer in supporting the software life-cycle. The current intent is to develop an ASDE that will provide an environment that incorporates all the functions of the classic environment along with integrated support for one or more of the software engineering methodologies. Therefore, the environment must perform the classic functions and it must invoke the discipline of a software engineering methodology, along with providing all the bookkeeping support required by the method.

ASDE's can be envisioned as a three dimensional space. The first dimension is the amount of automation that is provided the user of the system. Here, automation refers to the amount of work that the system takes on for the user in order to relieve the user of unnecessary burden. The second dimension is the amount of discipline imposed on the user of the environment as far as the underlying software engineering methodology is concerned. Finally, the third dimension is the amount of integration between the tools of the system.

Automation is very important in an ASDE because it is this ability of the system to do work for the user that makes the user desire to use the automated system instead of doing the work manually. The main feature of an ASDE is it's ability to assume the burden of the work in order to relieve the user of the onerous tasks that are unrelated to the solution of the problem.

> "This means that he can focus on the important issues and achieve a correct solution to his problem more quickly. Secondly, being freed from the responsibility of working out the details, he moves up to a higher conceptual plane, which is closer to the terms in which he conceives his problem" (Hammer and Ruth, 1980: 769).

Discipline is necessary in an ASDE so that the underlying methodology is adhered to. This is necessary if the ASDE is to aid in producing quality, well engineered software. The last dimension, integration, is also vital to this morphology because a fully integrated environment is easier to automate. Also, the amount of integration indicates the smoothness with which the user will flow through the system, thus providing less interruption of the thought processes.

At the present time, research into ASDE's is branching into two major paths. The first path is one in which the ASDE is a collection of tools which are defined by the system builders in order to support a specific software engineering method. The collection of tools is firmly fixed and can only be changed by a new system release. The advantage of such a

system is that the discipline may be firmly embedded and is hard to subvert. Also, the system is provided as one integrated environment and is standard in every organization. The disadvantage is that the discipline imposed may, or may not, stifle imagination and inventiveness. Another disadvantage is that the system may not fully meet the needs of the organization and because of the inherent inflexibility, it cannot be easily modified.

The second path of research leads to an ASDE which is basically a "tool kit" approach. In this type of environment, a minimum set of tools have been defined in such a manner that it is easily extensible. The working environment will, obviously, vary from organization to organization depending on the needs of the organization. This approach, which is a descendant of the classic environment, is more useful because of the flexibility, but will also be non-standard across implementations. Also, these types of environments tend to be composed of many small tools and the amount of integration will tend to be small. Therefore, the user's journey through the system will tend to be choppy and disjoint. Another drawback to the added flexibility is that the introduction of new tools to the system will make it easier for the user to subvert the discipline of the software engineering methodology that is being used.

The great amount of difference in the two approaches

highlights the complexity of the subject of ASDE's. This statement is substantiated by a noted expert on the subject of software development environments, Leon Osterweil of the University of Colorado.

> "The task of creating effective (development) environments is so difficult because it is tantamount to understanding the fundamental nature of the software process" (Osterweil, 1981: 35).

In fact, the optimal environment for most applications is found by extending the three dimensions of the software development space as far as possible. In addition, the environment concerns should include the concepts of user-friendliness, life-cycle support, consistency, traceability, explicitness, documentation capabilities, testability, and the capability of updating (Osterweil, 1981: 36-37).

1.3 Problem Statement

The Air Force software community has exhibited a great need for any tools that help to produce quality software at a reasonable price. Automated software development environments that embody the concepts of software engineering and life-cycle support are believed to be such tools. The objective of this thesis is to define the basic framework of an ASDE. After this basic design is completed, the key tool

in the system is identified. This key tool is defined as the tool around which a complete, fully integrated system can be constructed. The final phase of this thesis is to design and implement this key tool.

1.4 Thesis Scope

As implied in the background section of this chapter, the specification and development of an ASDE is a complicated and little understood process. This thesis investigation represents an initial attempt toward the realization of such a system. The specification and preliminary design of Sire, an ASDE, along with the detailed design and implementation of a key tool sets the stage and tone for the continued development of the Sire system.

1.5 Assumptions

The implementation of the Sire system is hosted on a VAX 11/780 computer under the UNIX[1] operating system. Therefore, full use of such items as system tools and the virtual architecture is implied. This choice of operating

---

1. UNIX is a registered Trademark of Bell Laboratories, Inc.

system, with resident tools, also drives the choice of the main implementation language towards the "C" programming language because of the ease of interface with the operating system that this language enjoys.

When implementing the key tool of the Sire system, the major concerns of the project were correctness, usefulness, ease of maintenance, ease of understanding, completeness, and on-time delivery. Therefore, such topics as optimization for time and space were only considered when they did not interfere with the main concerns of the project.

1.6 Approach

As with any software development project, or any research project, this effort began with an extensive search and review of current literature concerning software development. Of particular relevance were the fields of software engineering and software development environments. The main thrust of this literature review was to gain a thorough understanding of the software development process, the methodologies used to support the process, and to understand how the methodologies can be supported best by automated means.

The next step after initiating the literature review was to start the actual software development process for the Sire

project. As discussed earlier, the first stage of the general process is Analysis and Design. Refering to Figure 2, it can be seen that the first step is to begin the identification of requirements that are used in the remaining portions of the life-cycle. Many of the requirements are gathered from lessons learned in the literature review. The requirements analysis stage produced two important results. First, it provided a high-level model of the system to be developed, and it established a set of parameters and criteria against which the Sire system could be developed and evaluated.

The next stage of the Analysis and Design phase is the preliminary design stage. At this point, the project diverted from the normal software development life-cycle. The first concern during this stage was to develop a preliminary, high-level design of an ASDE. After this design was completed, it was evaluated to determine which part of the design was most critical. The critical part is defined as the part which has the most influence on the overall design of the system. This part, if designed and implemented correctly, will ease and drive the design and implementation of the rest of the system. In a normal system development process, the project would continue down the "waterfall" of the life-cycle. However, since the stated objective of this project is to fully develop only a portion of the Sire system then this is as far down the waterfall as this part of the project proceeded.

Instead, the next step was to start back in the Analysis and Design so that the entire life-cycle process may restart and the critical, or key, part of the Sire system could be correctly and fully developed. The development of this part spanned the entire life-cycle.

1.7 Summary

The software development process can be viewed as a very complicated process that is not very easy for one person to fully comprehend. Therefore, the use of computers is being explored in an effort to make the development process a little easier to use with the hope that "better", more correct programs are developed. The specific goals of this project are to define, design, and develop a tool, or sub-system, around which a full environment can be constructed. This is attempted by applying well-defined software engineering principles, as well as common sense, to the development of an environment that is geared to the human-oriented support of every phase of the life-cycle.

Chapter 2

Requirements Definition

2.1 Introduction

Requirements analysis is the process of defining the complete and explicit statement of the problem to be solved.

> "It focuses mainly on the interface between the tool and the people who need to use it. Other aspects - such as time, costs, error probability, chance of fraud or theft - must be considered among the basic requirements before an appropriate solution may be chosen. Requirements analysis can help understand both the problem and the tradeoffs among conflicting restraints, contributing thereby to the best solution" (Zelkowitz, 1979: 3-4).

Once all requirements are identified, it is common practice to attempt to compose a document that contains a complete, explicit, and unambiguous statement of the system requirements (Hadfield, 1982: 23).

This document, referred to as a Requirements Definition Document (RDD), should contain both a textual description of all requirements as well as a high-level model of the system. This model, usually in some graphical form, should exhibit all the requirements set forth in the textual description in both

an explicit and implicit manner. Upon completion, the RDD should provide a description of the system and the system's fundamental objectives. These two should be detailed enough to provide a set of parameters and criteria against which the system can be developed and evaluated.

2.2 Requirements Analysis

A major portion of this project was the review of current literature pertaining to the proper development of software. Therefore, the knowledge gained in this review had a great influence on the overall project. It probably had the greatest impact in the area of identifying the goals and concerns of the Sire system since many of these were taken from the goals and concerns of current research in similar projects.

Most often, requirements are presented in two levels. The first level usually consists of seven or fewer general requirements. These requirements apply to the entire project and are best thought of as the ideals, or inspiration, of the project. These ideals are specified at such a high level that no specific goals or requirements can be gleaned from them. They provide the guidelines for the second level of requirements.

As implied, the second level of requirements are the specific requirements. These are concrete enough that the system designers can use them to drive the design of the system and to provide detail. Note that the specific requirements may seem to belong to two or more different general requirements.

2.3 General Requirements

The definition and specification of general requirements for a project is a very important starting step. Therefore, the following requirements were carefully selected from a list of general requirements because of their relevance to the Sire project.

2.3.1 Reduction of User Burden

As mentioned in earlier sections, the main job of an ASDE is to support the user in the production of quality software. In doing this, the environment should try to aid the user by not only adding tools and other facilities, but it should make it easier for the user to perform the task of software production. "Reduction of the User Burden" is a term that indicates that the ASDE should provide the user with greatly expanded capabilities while making less work for the user.

2.3.2 Reduction of Software Errors

Although it is mentioned several places that the objective of any software project is to produce "quality" software, the term quality is never defined. It is intuitive to most people that the amount of errors in a program reflect negatively on the quality of the program. This is especially apparent when the potential cost involved with software errors is realized.

In the most extreme example, software errors may cost lives when the software in the flight program controlling an airplane fails. Simple software errors that occur frequently in the development of software may be responsible for costly budget overruns and schedule slippage. Persistent software errors cause operating problems for the end-user of the system and may cause problems such as bad operating decisions when the financial software that a company depends on fails.

Given that software errors are very costly, it should be obvious that their elimination is desirable. However, this is not such an easy task. One of the theorems of software engineering is that there are an infinite number of undetected errors in any software system. Therefore, all the developer can hope to do is to design the system in such a manner that all the detectable errors are discovered and corrected and that the undetectable errors never become detectable by

interfering with the operation of the system. This is one of the main reasons that software engineering methods were developed in the first place. Therefore, to provide the desired reduction in software errors, the ASDE must support good software engineering methods and practices.

2.3.3 Easy to Update

One thing that can always be counted on is the fact that once a project is finished, someone somewhere will want to change the software for some reason. The reason might be that there is an error, that the system needs extensions or that it needs to be adapted to a particular user organization to make it more responsive. Therefore, the system should be designed with emphasis given to the methods that must be followed to modify the system. This modification process should be easy to do and should require complete documentation so as to minimize the chance of errors being made in the modification process (Hadfield, 1982: 34).

2.3.4 Project Management Concerns

Software errors aren't the only reasons for project overruns and slippages. Many times the management of software projects make mistakes in scheduling, resource planning and other management concerns that cause these problems. Therefore, a complete ASDE should attempt to provide the

management with some type of control over the project without impeding the actual development process. Facilities that estimate current resource utilization, current project completeness, and current schedule impact are examples of some of the controls and tools that may be useful to management.

2.3.5 User-friendliness

User-friendliness is a frequently used term that has no concrete definition. Most people recognize the terms as meaning that the interface between the user and the system is natural and easy for the user to use. However, the experienced user of a system may think that it is user-friendly while the novice may think it is horrid. Generally, the most user-friendly systems are those that are easy to learn and once learned, are easy to use. Some systems provide one at the expense of the other because it is hard to provide both, just as it is hard to try and please everybody all the time. Even though it is hard to provide, user-friendliness is a worthy goal.

2.4 Specific Requirements

2.4.1 Automated Documentation Support

"One of the predominant underlying themes of discipline in software design and development is the need to commit all major steps and decisions to writing." (Wasserman, 1977: 354). This documentation will be used for many tasks, including further development, writing of user manuals, and maintenance. Although recognized as crucial, documentation is often overlooked because it consumes personnel resources and it is considered very tedious work. By automating as much of the process as possible, documentation will become less costly, less of a user burden and much more thoroughly done.

The environment should have the capability to produce an extensive variety of documentation. (Wasserman, 1981: 7). This includes graphical as well as textual forms of documentation. However, it must be realized that the environment cannot produce all the documentation automatically and that it can only act as an aid in some areas.

2.4.2 Flexibility

Flexibility is a key issue in the development of an ASDE. A useful environment must appeal to a broad base of users. This implies several things. First, the environment must support projects of different sizes. Second, it must be able to support projects of different types, such as

scientific, mathematical, real-time, etc. Finally, it must be usable by different users with different knowledge levels. The environment that contains the flexibility to do all this will appeal to the most users.

2.4.3 Integration

One of the major themes underlying a good environment is the support of the life-cycle concept. Usually a different tool is developed to support each part, or stage, of that cycle. A fundamental problem to date has been the fact that the tools are not compatible among themselves. (Wasserman, 1981: 5). This makes it harder to use the environment to its fullest capability. Therefore, the integration of all the tools in the environment is a very important requirement.

2.4.4 Language Independence

In the current software development world a number of programming languages are in use. Each language is best suited for different types of applications and there is no "best" language to use in all circumstances. The choice of language will depend on such factors as application, design, hardware environment, programmers experience, and so on. Therefore, many times a project will not choose an implementation language until some later stage in the life-cycle. This means that the ASDE must be somewhat

flexible in the area of programming languages. In fact, it is most desirable if a design can be run through the entire life-cycle and produced in different languages by flipping a switch. This gives the designers of the system the most flexibility to choose the language that is best for the circumstances.

2.4.5 Maintainable

Perhaps the key to continued success of any software product is the degree to which the software can be maintained. It is obviously not desirable for the system to be designed so that is is hard to change and correct errors. If this situation occurs then the useful life of the system will be shortened because errors will occur which, not being fixable, will render the system useless. Therefore it is critical that this system be designed for maintainability.

2.4.6 Open Ended

As discussed previously, an open ended environment is desirable in that it can be easily changed to meet different needs. Therefore, the system must be able to be easily modified without destroying integration and the underlying methodology.

2.4.7 Pre-fabricated Design

Many times, people find themselves re-inventing the wheel. In many cases this redundant effort is wasteful. Studies have shown that in the case of software projects, it is often the case that 40 to 60 percent of the modules being developed are already in existence and are available for use. (Lanergan, 1981: 297) Therefore, the ASDE should be able to reuse existing modules in much the same fashion that pre-fabricated houses are built. Also, the environment should be able to add to the base of existing modules.

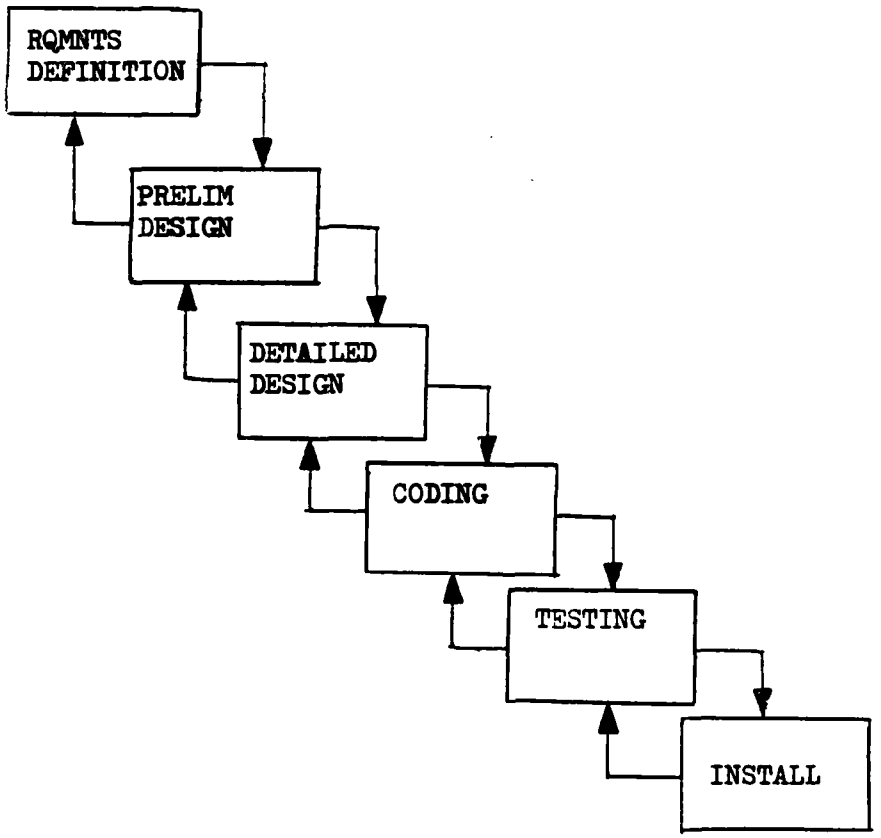


Figure 4. Life-cycle with Prototyping

2.4.8 Prototyping

"Prototyping has proven to be a valuable technique throughout the engineering sciences, but it has had little impact on the mainstream of software development." (Zelkowitz, 1982: 2). Prototyping is an approach to problem solving that uses the concept that the best way to see what is needed is to design a system and produce a crude working model in a very short time. This model is evaluated and the design is updated to reflect any changes. Then a new prototype is produced. This process goes on until the prototype is determined to be the solution to the problem. Hopefully, by doing this, the system designers/implementors will be able to more accurately determine system needs.

By adding prototyping into the life-cycle, the user can get feedback on the design so that requirements and design can be updated and improved. Adding prototyping into the life-cycle concept does change Figure 2 to indicate the feedback that is taking place. One of the possible representation of this change is shown in Figure 4. Prototyping also implies iterative development and necessitates ease of updating requirements and design.

2.4.9 Reliable

The reliability issue covers two areas. The first is reliability of the system and the second is reliability of the product. In both areas the reliability must be very high or the user will be tempted to use other, more reliable, environments or techniques. Reliability of the system means that the ASDE must perform as expected and not break or do unexpected things. Reliability of the product means that the ASDE must be able to produce a product that is reliable or, again, the user will not want to use the system to produce the software product. It is critical to note that the reliability of the product is partially a consequence of the design and the ASDE is not totally responsible for this issue.

2.5 Requirements Model

The second major part of the RDD is the high level model. This model may take several different forms, however, graphical methods seem to be the most favored because they can represent system architecture, design structure and software behavior. Also, they are very flexible and easy to understand. Understandably, some methods represent more than others.

Perhaps one of the most widely used graphical design methods is the data flow diagram (DFD) method. (Peters, 1980: 133).

> "The data flow diagram is used to partition a system and is the principal tool of analysis and the principal component of the structured specification. A DFD is a network representation of a system, and shows the active components of the system and the data interfaces between them." (Page-Jones, 1980: 51).

The DFD method is particularly useful in describing what is going on in the system without describing how it is being done. Also, the representation of the system presented by the DFD is hierarchical in nature. Therefore, the high level model will consist of several levels of detail. Because of these reasons, the DFD method was chosen to represent the high level model of the Sire system. The DFDs in the remainder of this chapter are accompanied with a textual description in order to assist the reader in understanding the breakdown of the model.

2.5.1 Sire Top Level

The top level of the Sire system in Figure 5 represents the user's perception of what the system is capable of doing. In this case, the system takes "system specifications" and, somehow, turns those specifications into a completely functional software system that is just what was ordered. This level of the model is purposefully vague in order to stimulate the imagination. Further detail, drawn partly from that imagination, is illustrated in the lower levels of the model.

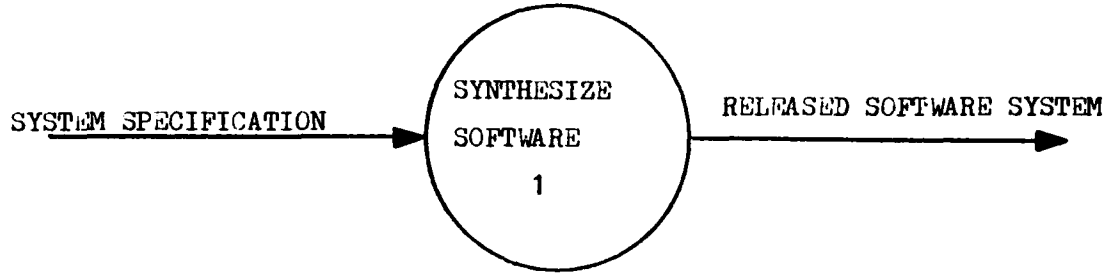


Figure 5. Sire Top Level

2.5.2 Synthesize Software

Breaking down the top level model of Figure 5 by "exploding" the Synthesize Software operation (operation 1.1) gives the more detailed view of the system depicted in Figure 6. This level of the model is strongly motivated by the modified life-cycle model of Figure 4. Easily identified from this level are the Requirements Definition, Preliminary Design and Detailed Design stages of the "waterfall". They correspond directly to operations 1.1, 1.2 and 1.4. Figure 7 shows how the other steps fit into this level of the model.

This model of the life-cycle is one that was followed in the design of the Sire system. Operation 1.3, Determine Project Status, is the only operation which does not fit into the life-cycle of Figure 7. It is included to help meet the requirements of providing Project Management Concerns, as detailed in the textual requirements defined earlier. Indeed, at many of the other levels of this model will be operations concerned with tracking the project status.

This representation of the system allows for designing and implementing a system by following completely through the software life-cycle. Also, it implies some sort of automated documentation support because of the Requirements Definition Document, Preliminary Design Document, and Detailed Design Document flows exiting from operations 1.1, 1.2 and 1.4.

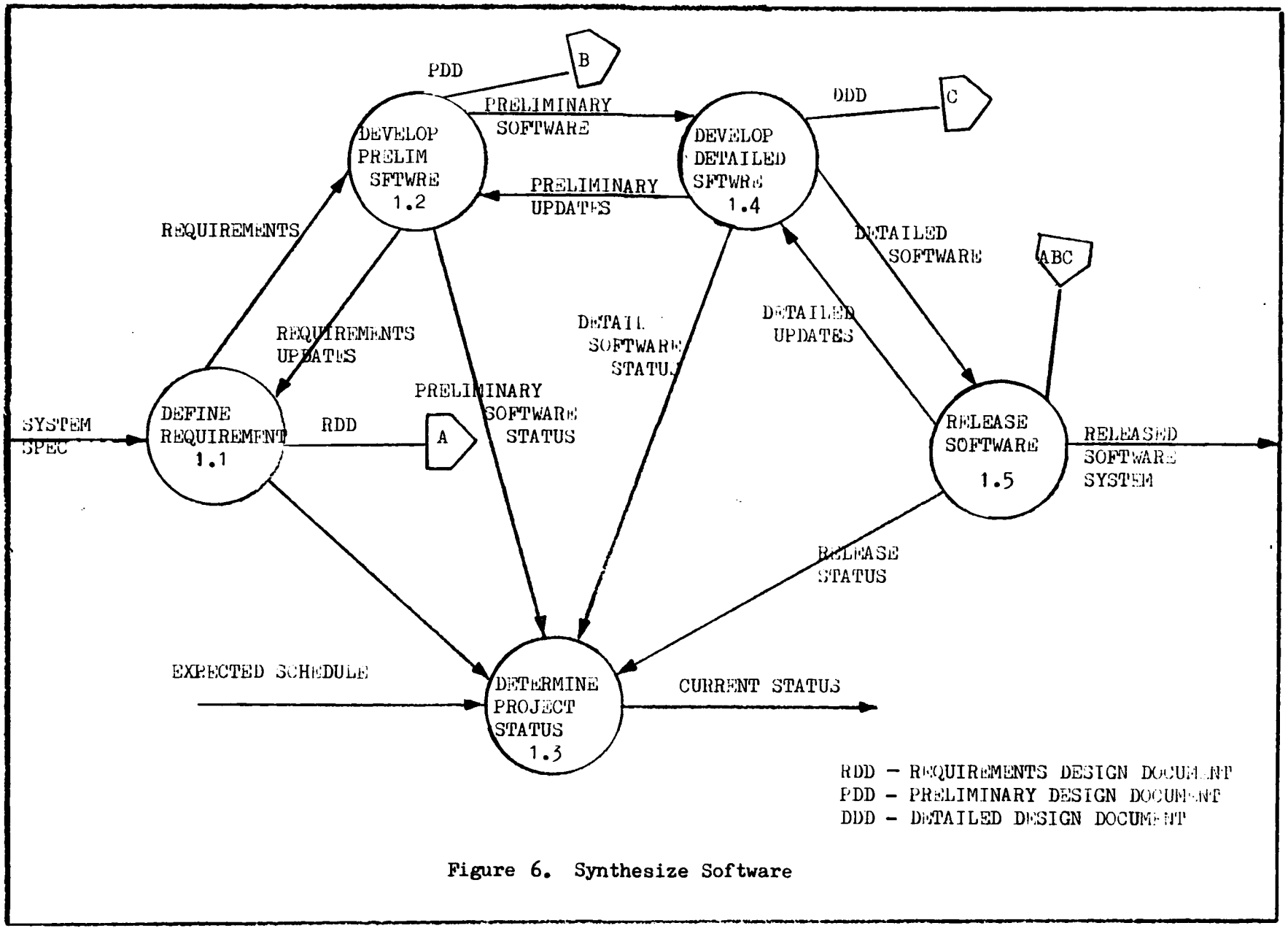


Figure 6. Synthesize Software

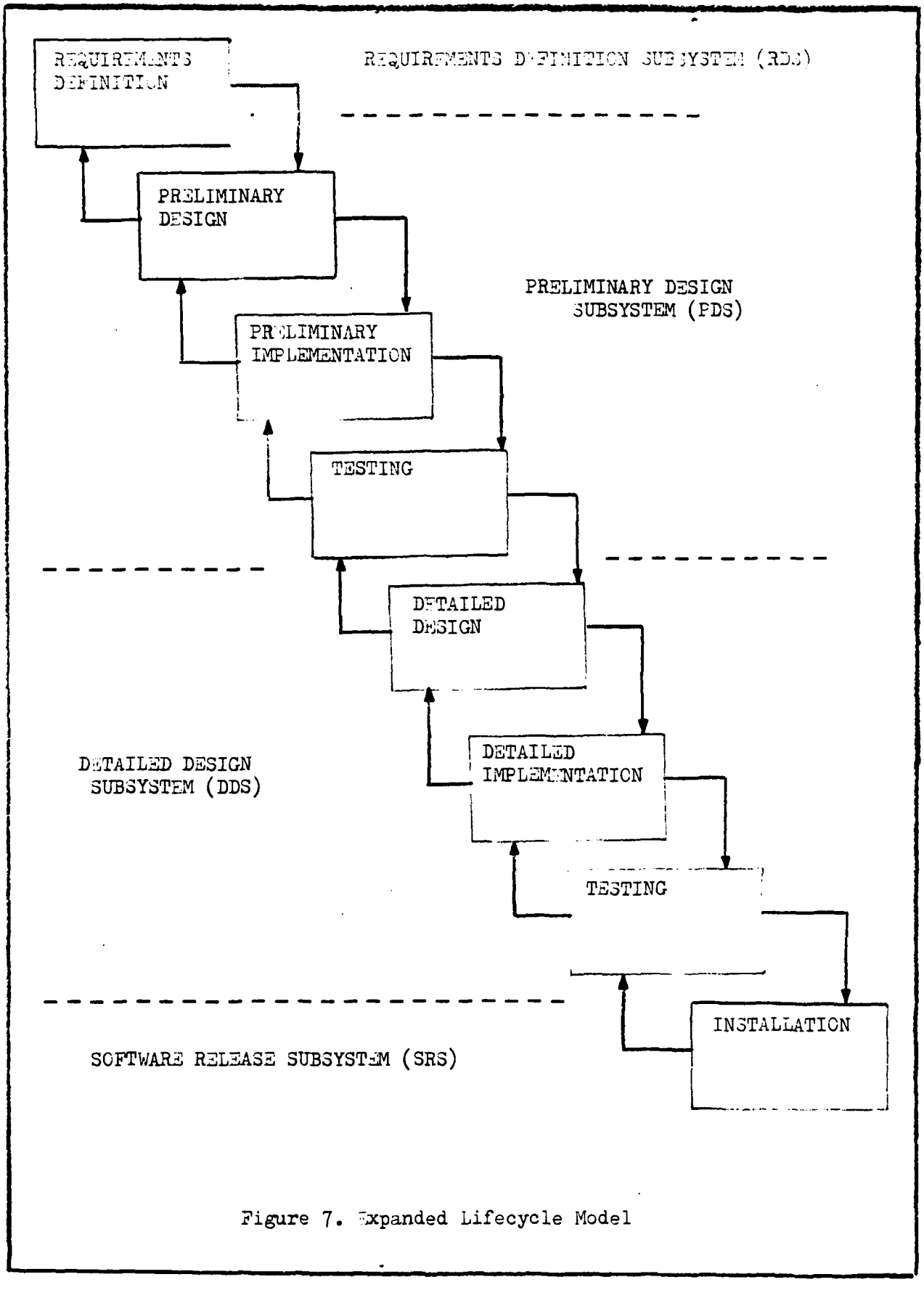


Figure 7. Expanded Lifecycle Model

2.5.3 Define Requirements

As mentioned previously, the Requirements Definition Document is composed mainly of the textual requirements and a graphical model. Therefore, it is only fitting that the breakdown of Define Requirements, represented in Figure 8, be concerned with these two tasks. From the System Specifications, textual requirements are iteratively developed, analyzed and updated in operations 1.1.1, 1.1.2 and 1.1.3. The output from these operations are used to develop the graphical model and the RDD. Once again, operation 1.1.7, Track Requirements Status, has been added for project managements concerns.

2.5.4 Develop Preliminary Software

Based on the requirements developed earlier, both textual and graphic, the process of developing the preliminary software, represented in Figure 9, is begun. The first part, corresponding to operations 1.2.1, 1.2.2 and 1.2.3, is to iteratively develop, test, and update the preliminary design. Note that the operation of updating the preliminary design will also cause requirements updates to ripple back to operation 1.1.

The second part is to develop the Preliminary Design Document (PDD) for inclusion in the system documentation.

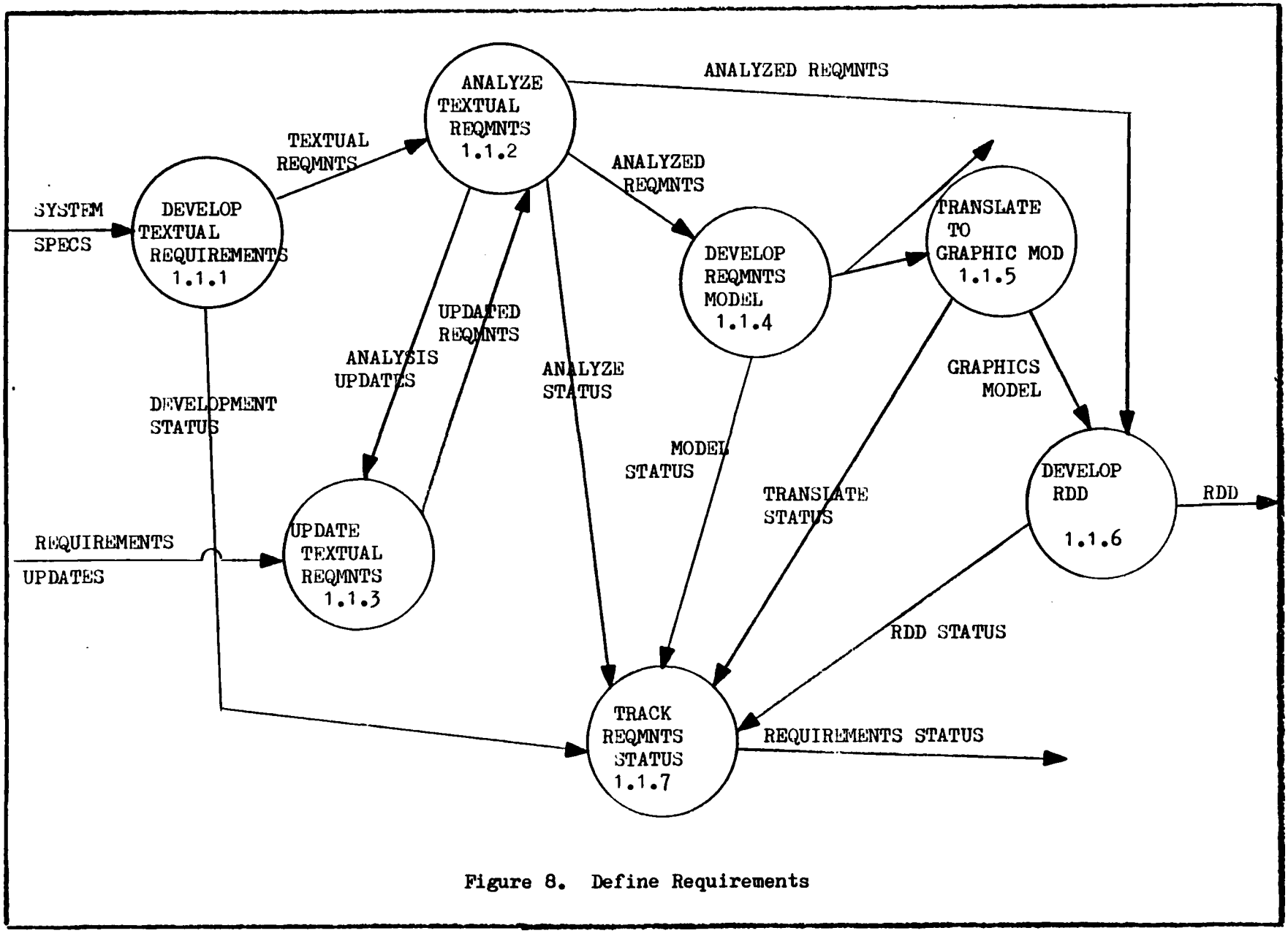


Figure 8. Define Requirements

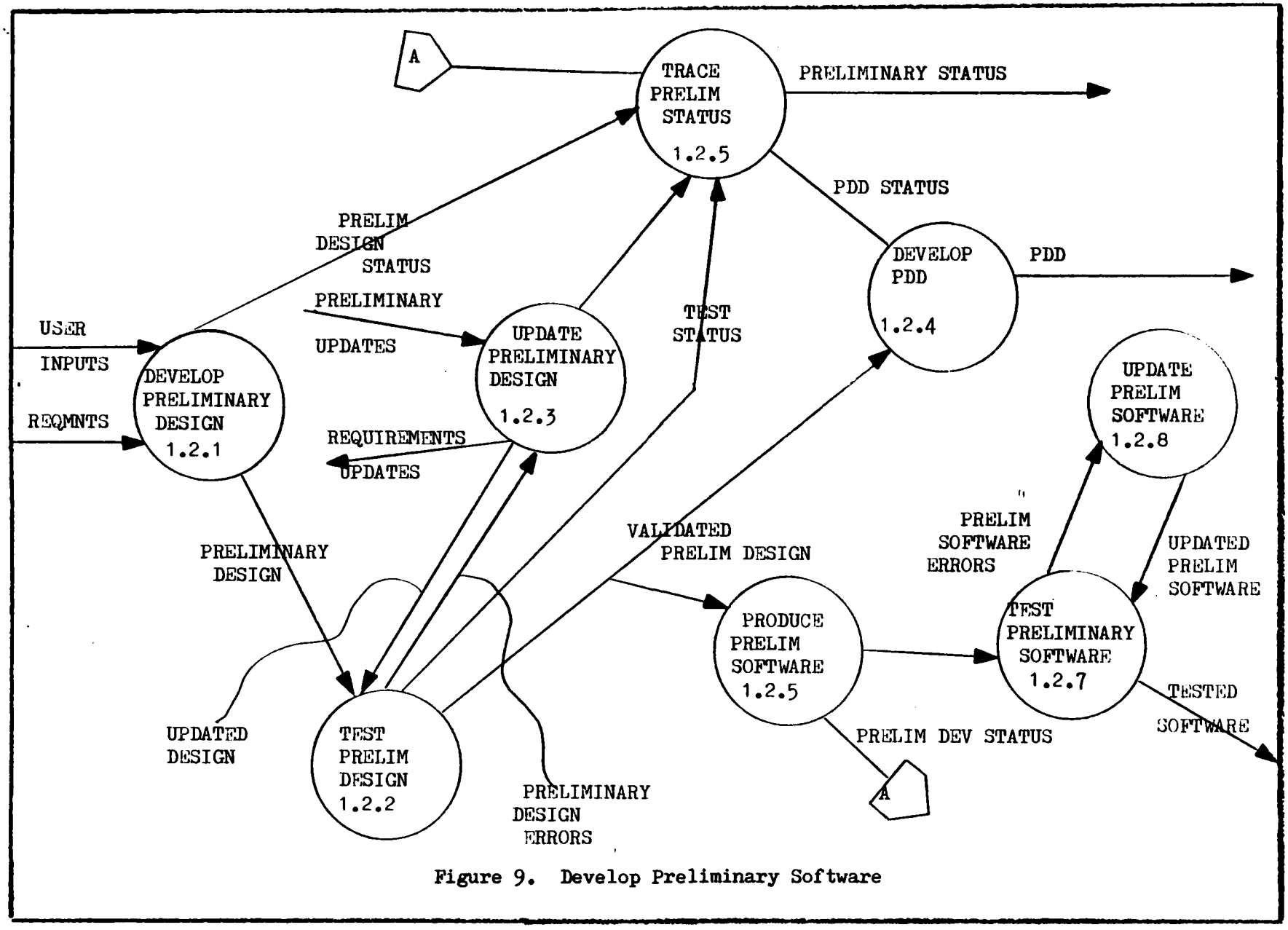


Figure 9. Develop Preliminary Software

This documentation process should be as automated as possible so that both human error and user burden are reduced. But then every part of the Sire environment should be as automated as possible.

The third part of developing the preliminary software is to actually produce the preliminary software. The result of the production, test and update cycle is a complete software shell into which the detailed software can be injected. This concept is a break from most software production theories in that some software is actually produced before the detailed design has been started. Another departure from traditional methods is that the selection of the language being used for implementation must come at this stage. The advantage of both of these points is that the output of the preliminary stage is a tangible software product that can be tested and evaluated. In other words, a stubbed prototype is produced and is available for analysis.

2.5.5 Develop Detailed Software

Based on much the same methods used in producing the preliminary software, Figure 10 shows the production of detailed software begins with the development of the detailed design based on the tested preliminary software delivered from operation 1.3 and the user inputs. After the design is developed then the detailed specifications for the software

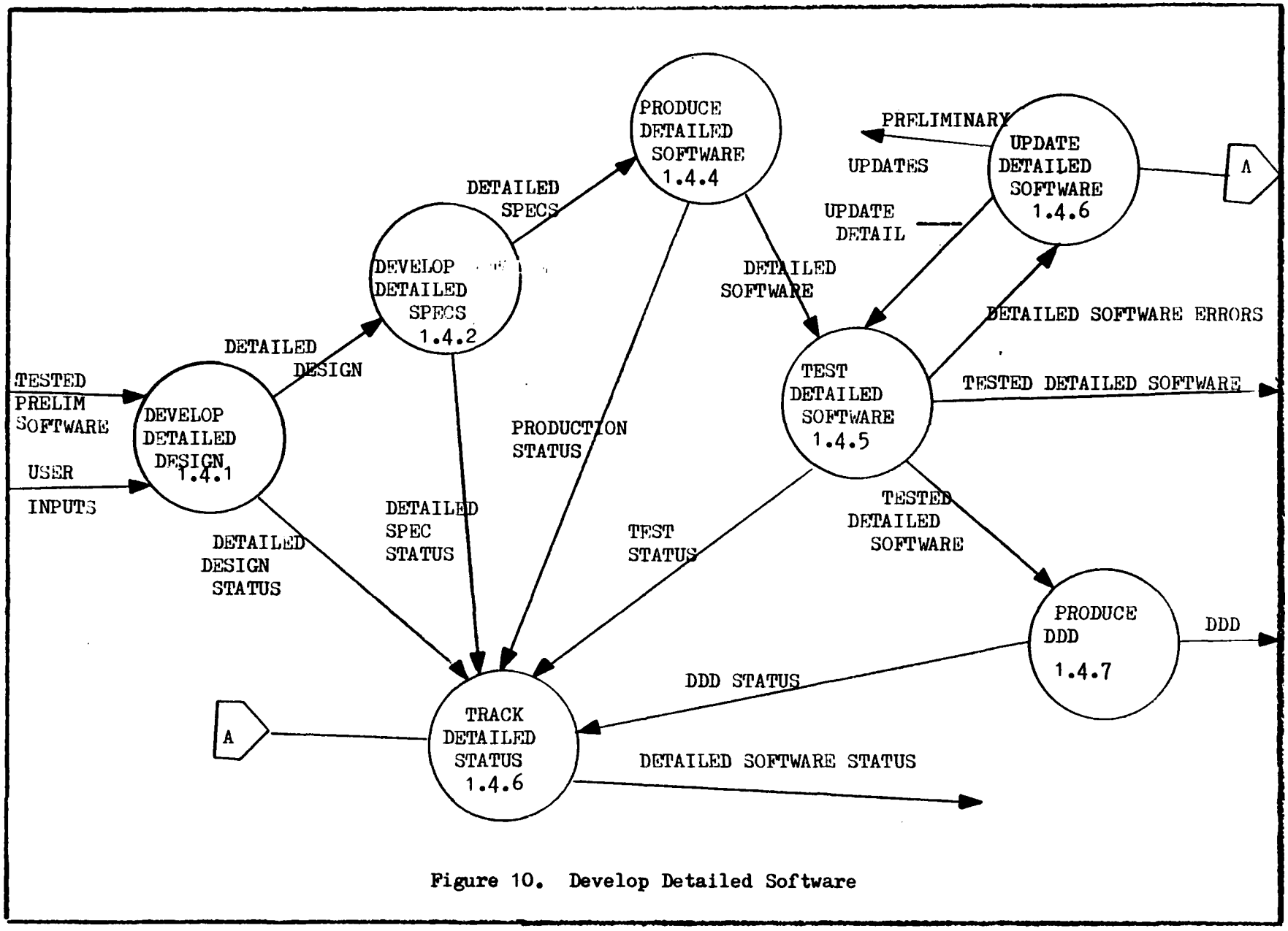


Figure 10. Develop Detailed Software

are developed. These specifications might take the form of a program design language or some other specification method. After the detailed specifications are complete then they are translated into the detailed software system.

2.5.6 Release Software

The final stage of the Sire high level model is the Release Software operation of Figure 11. This operation provides for the verification and validation, V&V, of the software system developed in the earlier stages. The V&V will provide any updates and modifications that need to be made to the system. The validated system is then used, along with the Requirements Definition Document, Preliminary Design Document, and Detailed Design Document developed earlier, to produce the system software documentation. After the system is documented it is packaged for release. This packaging may mean a range of things, from producing the documentation and transferring the software to a media for exchange if the product is commercial, to moving the system to a user area if the project is an in-house development project. This implies that the releasing of the software will be somewhat site/user dependent.

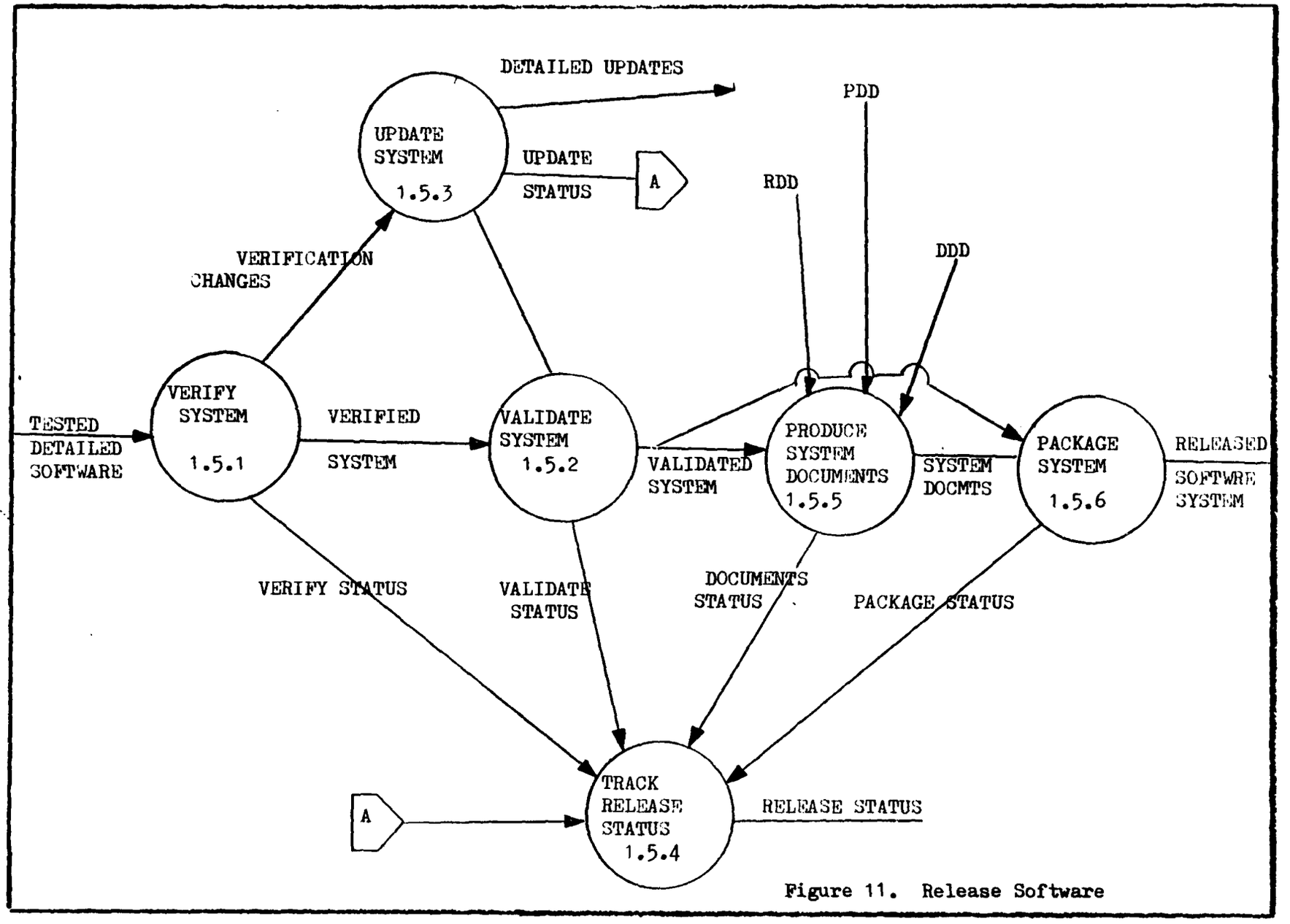


Figure 11. Release Software

2.6 Summary

The Sire ASDE specified in this chapter is to be a system that aids the software system designer in the task of producing software systems. The textual requirements specified are intended to drive the design of the system in the following stages of the life-cycle. Many of these requirements will explicitly appear in the design while others will be represented implicitly. Also, it is possible that not all requirements will be fully represented in the completed system due to possible conflicts between requirements.

Chapter 3

High Level Model Analysis

3.1 Introduction

With the completion of the requirements definition the next logical step would be to start defining how the system will work by beginning the preliminary design. However, the scope of this project does not allow for a complete solution to the system. Therefore, pausing from the logical progression for a time, an analysis of the high level model developed in the previous chapter is in order to determine which part of the model would be designed and implemented. This was an important step since choosing the part to be implemented would have great influence on future work done on the Sire system.

3.2 Model Analysis

3.2.1 Analysis Guidelines

Choosing a part of the model to be implemented was mostly a subjective process. However, there were several important guidelines to be considered. The first was that the implementation of the chosen part would set the standard for the rest of the system. This was not only a standard of quality, but also a standard of user support. However well the implemented part supports the user will greatly influence the amount of support the rest of the system will provide. This means that the implementation will cause the rest of the system to be implemented in a like manner using similar methods for accomplishing tasks.

3.2.2 Structure Analysis

The first step in the analysis of the model was to determine the parts of the system that were available for implementation. These parts may be either "point" parts or "distributed" parts. Point parts are sub-systems that exist in only one place, or part of the system. Distributed parts, on the other hand, are actual sub-systems that exist in a scattered out fashion in the system.

The most obvious point sub-systems appear in the model as operations 1.1, 1.2, 1.4 and 1.5. These parts will be known as the Requirements Definition Sub-system (RDS), the Preliminary

Design Sub-system (PDS), the Detailed Design Sub-system (DDS) and the Software Release Sub-system (SRS). In addition to these, the only other obvious point part will be the system driver, or Exec.

The only distributed part of the system appears to be the status tracking part. This part appears in many of the levels of the model and is concerned with tracking and reporting the system status. This part of the system will be called the Project Accounting and Reporting Sub-system (PARS).

3.2.3 Implementation Analysis

In examining the structure of the system, several conclusions about the part to be implemented seemed apparent. Stepping through these will lead to an obvious conclusion about the part which should be implemented.

The first conclusion was that each of the point sub-systems were completely self contained and therefore easy to implement as a part of the system. They all promise to be rather complex with the exception of Exec. Since Exec will, in all probability, be just a simple driver used to tie the parts together then it was eliminated from consideration. The next conclusion narrowed the field even more. Since PARS is a distributed sub-systems it would probably not be feasible to try and implement it without any of the other sub-systems being in existence. It should be fairly obvious that it would

be nearly impossible to test or evaluate a distributed system without any supporting code.

These two conclusions leave only the RDS, PDS, DDS and SRS parts to be considered. The SRS would appear to be in the same boat as PARS. Since the main purpose of this sub-system is to provide V&V and packaging, it seemed that these would be hard to provide for without some kind of product. Now, of the three remaining sub-systems it appeared that PDS is located in the center of the other two. It seemed that it would be the keystone of the operation since it's output drives the DDS and since it accepts input from the RDS. Therefore, a well designed PDS would drive the design and implementation of the rest of the system and it was the most likely candidate for implementation.

3.3 Summary

Resuming from this pause, the next step in this project was to narrow the scope to focus on designing and implementing the Preliminary Design Sub-system. This part of the model, chosen for it's central location and influence, was the object of further design, implementation and testing in order to provide a sound basis for further development of the Sire system.

Chapter 4

PDS Design

4.1 Introduction

After determining what needs to be done in the RDD section, the next step is to lay out the framework for the implementation of the PDS system. In constructing this framework, the high level model is changed from a requirements model into a functional model that represents the first level model of the preliminary design. There are many methods that have been developed to do this, among them are "transform analysis" and "transaction analysis" (Weinberg, 1978: 26). This not only provides the starting level of detail for the design task, it also helps assure that the requirements stated in the RDD are embedded in the system design. The purpose of this chapter is to briefly describe the preliminary and detailed design. The design presented here is that of the PDS sub-system. Appendix C contains more detailed information about the entire Sire system design.

4.2 Strategy

One important aspect of this project is that the overall, high level design of the system is entirely accomplished but that the implementation is only partial. This implies that, assuming the implementation will one day be completed, several different efforts must be concentrated on this system. This makes it important to design the system in such a way that as many requirements as possible are reflected in the design while not placing too many restrictions on the inventiveness and design capabilities of others working on the Sire system. Therefore, the design is kept simple and understandable. Another benefit will be gained from this approach since complexity is one of the major causes of unreliable software. Two concepts that are used to combat complexity, and therefore unreliability, are "independence" and "hierarchical structure" (Myers, 1976: 37).

The concept of independence is one that states that the independence of each component of the system must be maximized. This is generally done by partitioning the system in such a way that the interactions of the components is low. Independence is beneficial since an independent module is not influenced or controlled by others and it is not reliant on others to perform it's internal tasks. Complete independence

is not possible, however with greater independence there will be fewer interactions and those will be of less complexity.

Hierarchical structure allows the design to be represented as different levels. Each level represents the depth of detail of the system. Therefore, it also represents a level of understanding of the system. The benefit of the hierarchical structure is that it allows the user to define the organization and interactions without defining, or even understanding, the internal construction. This helps the designer work from a simple system concept to a detailed, complex concept in small, manageable steps, postponing the detail until more is known about the overall system design.

Independence also relates to an important concept of ASDEs. This concept is that of the "tool kit" approach discussed earlier. To some extent, the design of the PDS is influenced by this since effort was given to try and integrate proven, available tools into the system where ever possible. Therefore, integrating these tools into the system will add to the independence of the design since most of these tools are stand alone tools that have standard interfaces. Where tools are not available, effort was given to design as many parts of the system as if they were standard tools. This approach will increase the flexibility and ease the maintenance of the system by making it easier to swap tools for newer tools in much the same way that manufacturing assembly lines are

modernized by replacing obsolete tools and machines.

4.2.1 Design Representation Methodology

In order to represent the design work, Structure Charts are used. Structure Charts are especially useful for representing organization and for providing detail about the interactions shown in the DFD. Another benefit of Structure Chart methodology is that it is complimentary to the DFD methodology. By first representing the design of a system in DFDs, such techniques as Transform Analysis and Transaction Analysis are easily used to translate the DFDs into the high level Structure Charts. Therefore, because of their organization and ability to complement the DFDs, structure charts will aid in making sure all requirements that have been set forth have been met and that independence and hierarchical structure have been maximized. (Weinberg, 1976: 29).

4.3 Design

4.3.1 Motivation

> "Computer programming is, in many ways, like architecture. The programmer faced with a complex task must, like the architect, design a large object consisting of many parts that interact with each other" (Abrahams, 1975: 18).

Since many software engineers accept this as a basic premise,

then as many of the applicable methods of architecture must also be accepted. Chief among these methods is the blueprint. In fact, the use of the blueprint is the the very foundation of architecture. In designing a structure, architects use a blueprint, which is a series of specifications detailing the design of the structure. The blueprint is refined in successive levels until it results in a complete specification from which a structure can be built (Lewis, 1977: 226).

Applying this concept to software design is not very difficult. A specification language can be used to define the structure of the system under development. Using a top-down structured concept, this specification, or software blueprint, may be used to detail each level of detail of the software system. This leveled specification process, if implemented correctly, would be valuable in team programming efforts since its very structure would be useful in controlling and managing the project. The only hangup in this whole concept is that, to date, no single suitable specification language has been widely accepted. This is because not one has been recognized as being able to provide the same support to software development that blueprints provide to architecture.

Another benefit of specification languages is that, for the preliminary design process, the language may be simple, programming language independent, and may easily be translated

into the software shell desired in this implementation effort. Also, specification languages are very useful in providing a form of self-documentation for the system.

With all this in mind, the preliminary design of the PDS was based on the software blueprint concept. The specification language to used, MIDAS, was specifically designed for this project and is detailed in Appendix B.

4.4 Design Structure Charts

Although enlightening, a discussion of all the structure charts in the PDS design would become quickly boring and, in places, quite redundant. Therefore, several charts have been selected for their importance to the PDS implementation. These charts will be discussed in order to give some of the flavor of the design philosophy.

Main PDS Chart

The main thought behind the chart in Figure 12 is that, rather than build the MIDAS language description all at once, the description is built in stages. The three obvious stages are hierarchy, MID (module interface description) and data

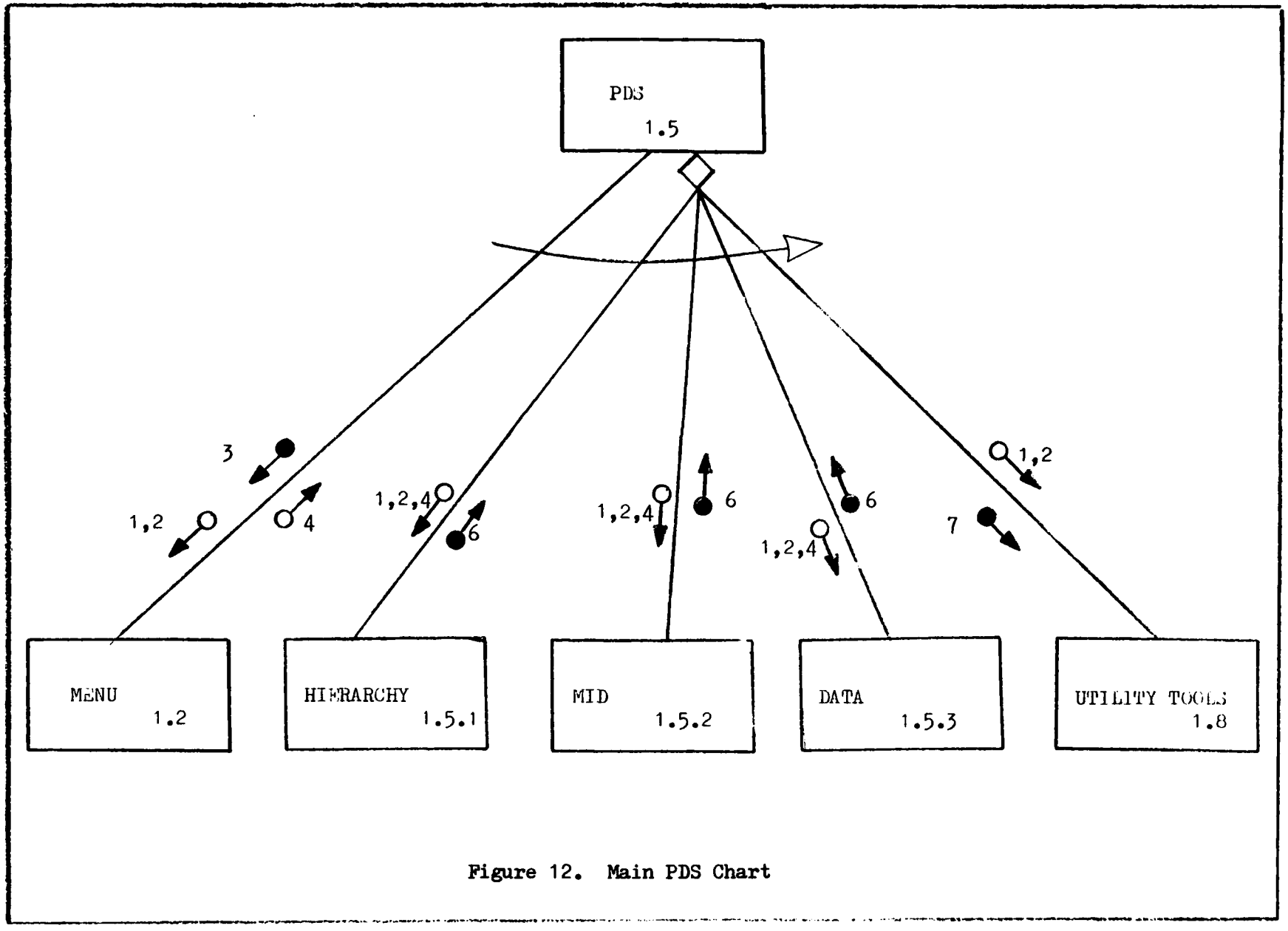


Figure 12. Main PDS Chart

description files. This method of operation allows the user to input the hierarchy, have it checked and then allow the system to work on it and generate a partial MID. The MID is then completed by the user and compiled, with the output being a partial data description. The data description is filled out and compiled. Then, if all conditions are right, the compilation of the data section will cause the generation of source code in the implementation language that has been chosen.

Edit/Compile Cycle

The edit compile cycle illustrated in Figure 13 is used in several places and is important to the PDS. The concept is very simple in that the user first edits a description file, compiles it, receives the error message and corrects the file. This continues till the compile function terminates normally. Upon termination, the compiler will have generated the next file to be edited. In this case, it will create a partial MID file.

This is really just a traditional approach to editing and compiling. However, tradition should not influence the manner in which this edit/compile cycle is implemented. For example, instead of the implementation having the tasks occur serially,

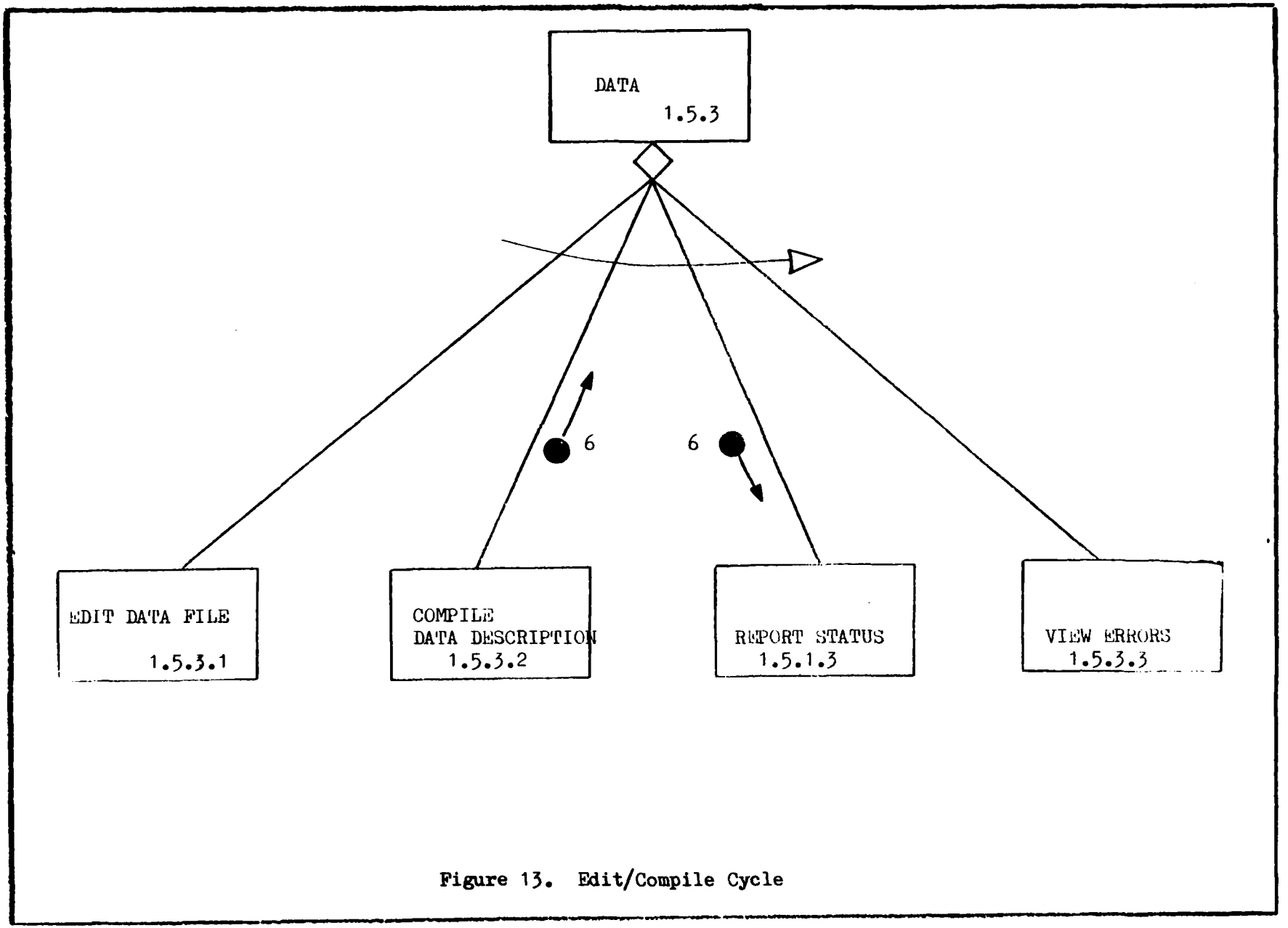


Figure 13. Edit/Compile Cycle

they might occur in parallel. A context directed editor might be used that compiles the information as it is entered and gives instant error messages. It is important to remember that structure charts describe the functions that need to take place and do not indicate when they should take place.

Compile

The chart represented in Figure 14 is here not because of its complexity, but because it explains what is meant by compile. When the word "compile" is used it means that the input file will be parsed, checked and an output file will be generated. The difference between this compilation process and the traditional process is that the compilers used here deal with a very small, simple, high level language. The code generated will also be a high level language that is not very far removed in complexity from the input source, unlike a traditional compiler which usually has a great complexity gap between the input and output.

4.5 Summary

The goal of any design should be to try and produce a

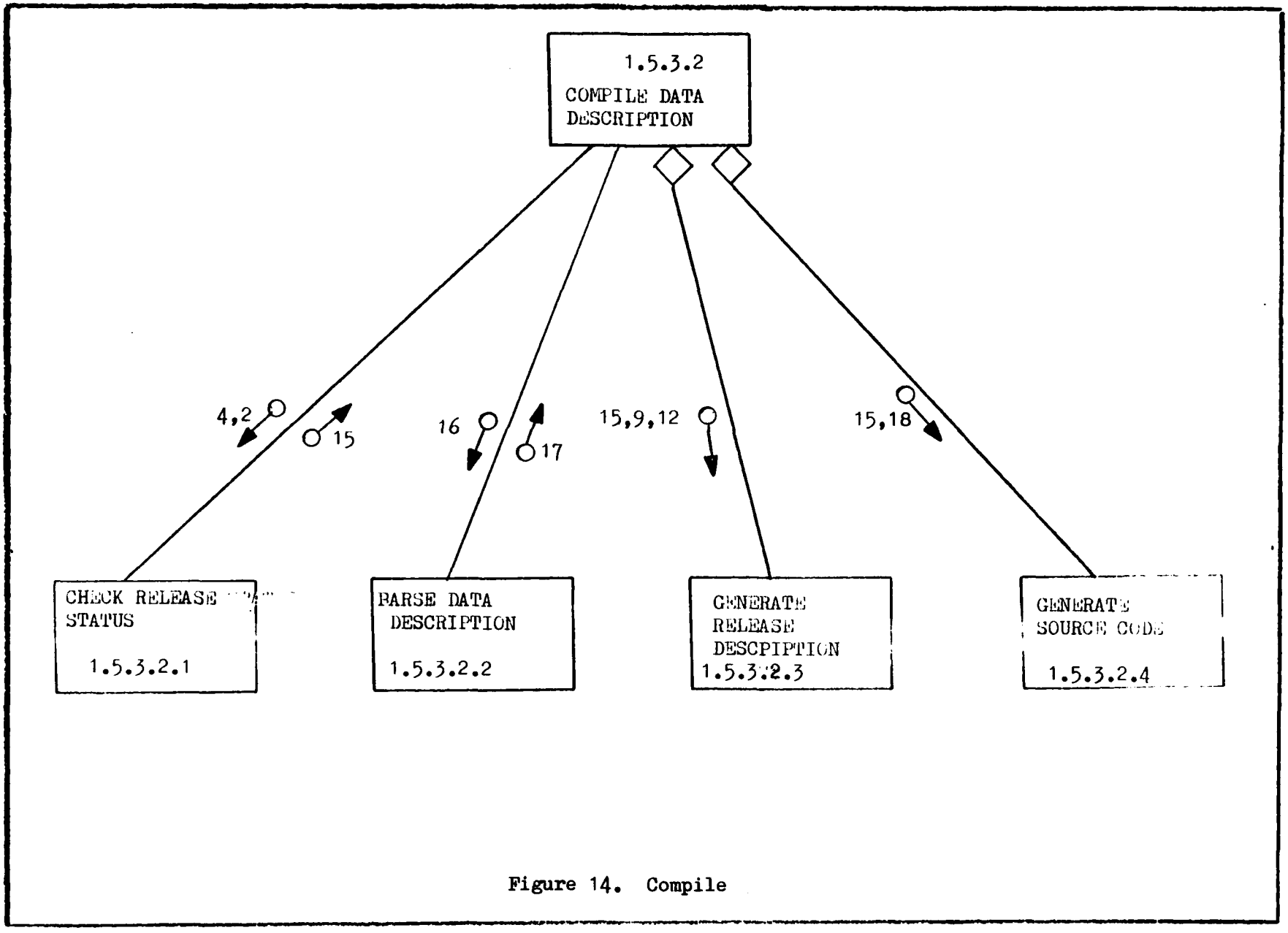


Figure 14. Compile

flexible, simple and functional design. Although this seems obvious, it is hard to do. The principles of software engineering should be followed and simplicity and common sense should rule. The resultant design should appear to be obvious and should be disappointing because it seems too simple. Hopefully, the PDS design fits this description. It seems simple and easy to implement. Whether this is an indication of good design or simple-mindedness remains to be seen.

Chapter 5

Implementation

5.1 Introduction

"There is an almost infinite number of ways to implement and test any system. Indeed, there is an almost infinite number of organized approaches to implementation and testing!" (Yourdon, 1978: 376). The method of implementation usually is driven by the requirements and special considerations of the project. Because of the partial implementation strategy of Sire, the approach used in this implementation is related to the design strategy used. In other words, it is basically a top-down approach. It is also a depth-first, left to right approach. This means that, when viewing the structure charts, the implementation start on the left side and proceed down the left branch until that left branch is implemented. Then, the implementation moves to the next branch. This scheme differs from others in that it is top-down, but not strictly top down. Also, because of the structure of the design, some parts of the system have been totally implemented before the others were started.

This approach is beneficial because it almost eliminates the need for a comprehensive, detailed system test plan by replacing it with incremental testing. This incremental testing occurs as each increment, or portion, of the design is implemented (Yourdon, 1981: 381). This method allows for "user-feedback" sooner in the project. This is helpful in correcting any mistakes or deficiencies, but also makes the implementation stage drag on longer than it would if a traditional approach were to be used.

5.2 Implementation Strategy

Once an approach to implementation has been chosen, the next logical step towards the implementation is to develop an implementation strategy. This strategy is dependent upon the nature of the design and the desires of the implementor(s). In this case, the strategy was to decide first upon an implementation environment and language, then to decide upon which practices would be used in the implementation.

5.2.1 Environment and Language

The choice of both an environment and a language are very important. The environment must be capable of supporting the final implementation at a minimum. It would be more helpful if, during the implementation, the environment actually aided

the implementor(s) in producing a quality system. Going hand-in-hand with this concept is the implementation language. It also must be able to support the development of the system and should aid in the production. In addition, the language should also be suitable in as far as maintenance and readability are concerned. A well structured and readable language will not only aid implementation, but will aid future attempts at maintenance and modification.

It should be obvious that the idealistic views concerning the choice of an environment and language need, in actual practice, seldom be worried about. Often, there is no choice about the environment to be used because there will only be one available. As far as the language is concerned, there are often only one or two suitable languages available on the environment.

As far as the Sire system is concerned, one of the basic assumptions set forth in the beginning was that all implementation would be done on a VAX 11/780[1] with a UNIX operating system. Therefore, with the environment chosen, the only thing left would be to choose a suitable programming language.

In choosing a programming language, several factors had

---

1. VAX 11/780 is a trademark of Digital Equipment Corporation

to be considered. First of all, the language must be able to support the development of the system. Second, it must be readable and easy to use and maintain. Third, it must support modern software engineering practices. There are also other minor considerations that must be weighed when making the final choice. Chief among these is the consideration that the implementor(s) be able to use the language to its fullest extent without having to constantly be struggling to learn a new language. This consideration is often overlooked as new languages are often thrust on implementor(s), resulting in a slow down of the implementation and a less than efficient product.

Given these factors and considerations it was not difficult to choose an implementation language for the Sire project. The chosen language was "C". The main reason for the choice is that in a UNIX environment, there are no languages more powerful or flexible. This is because the C language was used to implement the UNIX operating system. As the system development language, there is a very close tie between UNIX and C which makes available a great many system tools to the C language implementor. The great amount of operating system support given the language, the number of tools available, the flexibility of C, and the fact that the Sire system designer and implementor is familiar with C, made it an ideal choice for the implementation of the PDS system.

5.2.2 Practices

Since one of the factors used in choosing the programming language was the desire to use good software engineering practices, then it is only appropriate that some of the major practices to be used in the implementation be discussed at this time.

Programming Methodology

The programming methodology used in this implementation has already been identified as the top-down approach. This was important to this implementation because it not only provided earlier testing and feedback, but it also made the partitioning of the implementation into separate, stand alone modules easier. This concept is very similar to that of separate compilation, which is available in C and was heavily used, in that different parts of the system may be implemented without the others existing.

The major difference is that each part, when completed, will be a completely functional software program. Therefore, the final step of the implementation was to just gather all the separate parts together and make sure that they interface

and interact correctly. This approach goes back to the tool-kit approach discussed earlier. This not only make it easier to see results, but it will allowed for these parts to be modified and maintained separately.

Programming Style

It is now widely recognized that if a program, or system, is going to be used and maintained for some time, then it will probably be read much more often than it is written (Sommerville, 1982: 117). Therefore it is very important for the program to be as readable as possible. Readability was previously linked to the programming language used. However, even if a readable language is used it is still very possible that the resulting code will be very unreadable. Readability, given a good language, becomes very dependent on the programmer. The C language has been criticized because it is very easy to produce very cryptic code. However, with care and thought, the resulting code can be very readable. With this in mind, all steps possible were taken to maximize the readability of the program code. These steps included structured programming concepts, the use of extensive program documentation and the selection of relevant variable and function names.

Programming Tools

"One of the most important developments in the practice of programming has been the realization that the programming process can be supported by a number of software tools" (Sommerville, 1982: 128). The concept of program development tools is widely supported by UNIX, which is recognized as having a very extensive and varied set of tools. Sometimes these tools aid the production of code, sometimes they produce code (code generators), and sometimes they provide utilities that the implementor can take advantage of, therefore relieving the implementor of the task of implementing a new utility. No matter what the tool does, it has some impact on the system.

Compiler

The most obvious tool used was the C compiler. The compiler obviously affects the implementation since it limits the programming language. The compiler used in this implementation was the standard C compiler provided with UNIX version 4.1 bsd.

AR

Another tool of importance to this project is the system utility "AR". This utility is a file librarian. It can be used to catalog and "shelve" files into one physical file. This tool is not only used by the implementation to maintain file libraries, but it was used as a code control system to maintain backups during the implementation process.

YACC

Besides the compiler, the tool which has the biggest impact on the implementation is YACC (short for Yet Another Compiler Compiler). YACC is a program that, given a simple syntactic description, will produce a complete parser for that syntax. It is also possible to include actions in the syntax description that YACC will cause to happen at the correct time. With this tool, the process of building the compilers shown in the design was cut down significantly. This tool added greatly to productivity, however, this increased productivity is not without cost.

YACC, by nature, is a very complex tool that produces complex C source code. While C can be a readable language,

YACC tends to produce cryptic code that is hard to understand and fairly slow. Also, YACC precludes the possibility of dealing interactively with the user. This lack of interaction caused major problems with the user-interface that had been planned. The system that resulted because of YACC was a little more awkward and less user-friendly than planned. These problems will be discussed in greater detail at a later time.

Lex

A tool that is complementary with YACC is Lex. Lex interfaces with YACC and is used by YACC to scan a file and provide tokens to the parser. Again, this tool was a great aid to productivity since it kept the implementor from having to write a separate lexical analyzer. The only bad effect that Lex has is that, like YACC, it has a tendency to produce large amounts of inefficient C code.

Make

Make is a system utility that helps to maintain system configuration by maintaining the correspondence between the source code and the object code of the Sire system. Given a

list of all the dependencies in the system, Make can, when invoked, make sure that the system is fully up to date. If it is not up to date then Make knows how to bring it up to date. Make is used not only to maintain the implementation configuration, but is used by the system itself in keeping the PDS produced code up to date.

Program portability

Program portability is an important topic. It is important because the more portable a program is, the more widely it will be used, therefore resulting in greater profits and benefits. The UNIX system offers a large amount of portability between different types of computers, especially for any C source code. However, this portability is not guaranteed just because UNIX and C are used, instead, the implementor must work hard, and long, to maintain this portability. Because of time constraints, portability was not a topic of consideration in implementing this project. Even so, the resulting system should be fairly portable with only one thing possibly getting in the way. That thing is the amount of dependence that the system has on the machine architecture.

Machine Architecture Dependencies

Although good design techniques do not usually take machine architecture into account, the implementation must. The machine that underlies the operating system may impose limits on aspects such as memory used, processing time, file space or other resources. The implementor must therefore take these into consideration. To provide portability, the implementor must look at all possible target systems and work with those in mind.

Since it has been stated that portability is not a topic, the system was implemented with the VAX architecture in mind. Of particular importance is the virtual memory scheme used by the VAX. The PDS implementation does all calculations and builds all structures in memory without consideration for size. This means that, if this implementation was transferred to a PDP-11[2] running UNIX, it is entirely possible that the PDP-11 would not have a large enough memory allocation scheme to support it.

---

2. PDP-11 is a trademark of Digital Equipment Corp.

5.3 Summary

The implementation of PDS can best be described as top-down and very structured. As discussed previously, the implementation followed good software engineering rules in general and the tool-kit approach to software environments in particular. Particular attention was given to making the system as maintainable as possible.

Chapter 6

Critical Analysis of Sire

6.1 Introduction

After a project is near completion it is useful to step back and take an objective look at what has been accomplished (if possible!). This analysis should concentrate on the strengths and weaknesses of both the design and the implementation. Also, the analysis should take place at a fairly high, conceptual, level and avoid the nitty-gritty. This analysis process is helpful in producing conclusions and recommendations about the subject matter.

This analysis will be broken down into three major parts. The first part will be an analysis of the design. Secondly will be an analysis of the prototype PDS system as implemented. Finally will come a section on how both the design and implementation helped fulfill the requirements set forth in the beginning of the project. After this part will come a summary, or list of some of the lessons that have been learned during this project.

6.2 Design Analysis

6.2.1 Weaknesses

The major weaknesses of the design of Sire seem to be sins of omission and lack of detail in some critical areas. These omissions and sketchy detail cause decisions to be made that may be detrimental to the overall quality of the system. specifically, detail is lacking in the PARS subsystem, the RDS subsystem and in the description and design of the system databases and data tables.

For example, the PARS subsystem is a distributed subsystem that will eventually have parts of itself spread out through the whole system. This is necessary for it to gain accounting and status information. However, there is no description of how it will do this, or even any description of the information that is needed. This leads to the rest of the subsystems being designed without the PARS modules in mind. This is questionable practice at best and will probably end up in conflicts when the PARS subsystem is to be implemented. This kind of thing often leads to what is known as a kludge.

6.2.2 Strengths

The strongest asset of the design is the inherent flexibility and top-down nature. Care was taken to use good software engineering techniques in building up the design. Therefore it proved to be very flexible and easy to use. Another benefit is that the design is very simple and easy to understand. This is really about all that can be asked for out of a design.

6.3 Implementation

6.3.1 Weaknesses

It often turns out to be the case that the biggest critics of a software system are the designers and implementors. This may be true because they know all the problems that exist in the system. The implementation of the PDS subsystem is one of those software programs. Although the basis of the implementation is sound, there are several problems that detract from its usefulness. These problems deal with the databases, the efficiency and the edit/compilation cycle.

As discussed before, the databases and datatables were not considered thoroughly in the design phase. Therefore, the implementation phase saw the willy-nilly creation and use of databases. The problem with this is that it probably caused

extra, redundant work since a thorough definition would have led to a single management scheme instead of the five or so that ended up in existence. Also, it is probable that instead of several small data stores, a single, formal database could have been defined that would have been of greater benefit to the whole Sire project.

The efficiency of the implementation has been discussed briefly in several places. Suffice it to say that the methods used were chosen for their convenience in implementation and not their efficiency. This led to the implementation being much too slow in several critical places. The most noticeable of these places is in the presentation of menus and error messages. Inefficiency in these places negatively affected the user-friendliness that was desired of the system.

Another, unfriendly, awkward part of the implementation is the edit/compile cycle. The user of the system finds it necessary to edit the source file, then compile the source, view the error message, look at the error listing and proceed as needed. This is very awkward since is is not the least interactive. This is mostly due to the use of YACC to produce the compiler sections and the use of a relatively unsophisticated editor. Ideally the best way to handle this would be to use a context sensitive editing system that compiled the source as it is being entered. This would provide for instant feedback and would save much time since

the four separate steps would be collapsed into one.

6.3.2 Strengths

The biggest strength of the implementation is that it is faithful to the design and that it correctly does what it was planned to do. That is, it provides a demonstration of the specification concept and how it can be used for automatic documentation and automatic generation of well designed source code.

6.4 Requirements Resolution

In the beginning of this project, it was necessary to decide what requirements were important. These requirements were put into two classes, the general and the specific. One way of measuring the success, or completeness, of this project is to decide how the requirements were resolved. This will lead to yet other insights as to how the design and implementation may be improved in the future.

It is important to realize that many of these requirements cannot be connected with any specific chart from the design structure charts or any piece of code in the implementation. Rather, some of the requirements are resolved by the general nature and procedures of Sire. Also, since the

design wasn't taken to it's lowest levels everywhere and since the implementation is not complete, all that can be done is to describe in which section the requirements should be met.

6.4.1 General Requirements

As noted above, the following general requirements, since they are general in nature, are more related to the philosophy and nature of Sire than they are to any specific part. The discussion of these requirements resolutions is not intended to be thorough, rather it is intended to give the basic flavor of the way in which certain requirements have been met (or not met, as the case may be).

Reduction of User Burden

The philosophy of Sire is that the user should only have to input information that the system itself cannot ascertain. For example, in the PDS section the user is still required to do some "coding" but, it is also true that much of the MIDAS description is filled out by the system itself, thus minimizing the amount of work for the user. Other major areas of user support are the automated documentation features and the ability of Sire to translate the MIDAS description into a source program shell for the user.

Reduction of Software Errors

The resolution to this requirement is hard to pinpoint and quantify with Sire being unfinished. However, it should be apparent that the desire of Sire to minimize the amount of work that the user must do will help reduce the amount of errors. Sire will probably be best at reducing syntax and coding errors, thus leading to faster production of software. Sire also intends to reduce logic and design errors through the use of rapid prototyping and the use of pre-fabricated software modules.

Easy to Update

It is questionable whether Sire can claim to meet this requirement or not. At this point in the implementation it does not seem that any solid evidence can be presented to show how Sire does, or will, meet this requirement. In fact, it is estimated that the structure and discipline of Sire may prove to make the system produced by Sire harder to update.

Project Management Concerns

This topic has already been discussed as a weakness of Sire. It seems that not enough attention was applied to this requirement in the design stage, therefore, there is no evidence, other than a vague design section, as to how Sire will support management concerns. This resolution of this requirement certainly warrants future investigation.

User-Friendliness

This requirement, though very vague in nature, was attacked by trying to be as nice and informative to the Sire user as is possible. Some examples of this are the use of menu-driven software, informative error messages, and use of screen attributes to present information to the user. After using Sire, it also becomes apparent that the friendliness could be improved through the use of more communication with the user. At times it seems that Sire just sits there going about it's tasks without informing the user of what is happening.

6.4.2 Specific Requirements

The following specific requirements are more specific in nature than those presented above, and in many cases they have evolved from the general requirements. Again, the discussion of the method of resolution for any requirement is not intended to be detailed.

Automated Documentation Support

The support of automated documentation is readily seen in the design structure charts. It is also apparent in the implementation of the PDS subsystem. The implementation automatically produces documentation that can be used as module description both in the source code and in the documents describing the code. In fact, the PDS system was used to produce some of the module documentation that appears in Appendix C. This is just one small example of how Sire can support documentation. Had the amount of time for implementation been greater such features as automated production of structure charts, automated data dictionaries and automated production of a preliminary design document could, and would, have been easily implemented in the PDS subsystem since almost all the necessary information is already available.

Flexibility

At this stage of development, Sire remains a very flexible system. There is nothing that would limit it to one type of project or another. This designed in flexibility may prove the undoing of Sire because in trying to be all things to all people, it may become unusable or undesirable to use. The key to this flexibility will be in the hands of the future Sire implementors since it will be easy for them to turn Sire to a different direction.

Integration

The lack of integration in Sire has already been discussed earlier when it was said that the edit/compile cycle in Sire tended to be awkward. This awkwardness is caused from trying to integrate poorly designed tools into Sire. They are poorly designed in the sense that they are not designed to blend with the rest of the tools in Sire. This lack of integration was an implementation decision caused by time constraints and is not caused by poor design or lack of foresight. It was decided to use as many available tools as possible in order to get a more complete implementation.

Language Independence

Although the present partial implementation is not language independent, it is only so because time was not available to produce more translation modules for the PDS subsystem. The basic design and philosophy of Sire provide for specification and design of a software system in a language independent manner. Then Sire takes over and, using translation modules, translates the design into working software. The only restriction on this independence is the number of translation modules that are available.

Maintainable

The subject of designing software for maintainability is not very well defined at this point in time. The best that can be done is to use good software engineering techniques and provide extensive documentation of the source code. This project was undertaken with this in mind. As a result, Sire is a relatively well documented software system that has been implemented in a readable high-level language.

Open Ended

Basically, Sire consists of two parts, the basic environment and the utility tools. The basic environment may be thought of as the PARS, RDS, PDS, DDS and SRS subsystems. These parts form the basic environment and do not meet the requirement of open endedness very well. Granted, the basic environment is very modular and can be modified by swapping modules in and out, however, the basic environment is limited to the task set forth in the design of Sire.

The utility tools, on the other hand, are very open ended. There are no specific design limitations on the tools and no specific requirements for them. Therefore, the tool list can be added to and taken from as need be. This concept of a separate tool kit allows the overall Sire environment to be tailored to fit the task. Also, the utility tools subsystem is extremely modular and easy to modify, thus making it even easier to tailor the tool kit.

Pre-fabricated Design

This requirement is resolved in the design of Sire. The design calls for a database of pre-programmed software modules

to be available. This database is also required to be extendable so that the database can be updated and added to if necessary. Although the design of the PDS called for an interface with this database of pre-fabricated modules, this feature was not implemented because of time constraints.

Prototyping

Prototyping is handled in Sire by the ability of the environment to rapidly produce, through translation, a skeleton software system at any time. This prototyping is not built in Sire as a menu feature, but must be managed by the system users. For example, the users could decide that they need a prototype system and they would proceed to specify and produce that system using Sire. Since Sire is designed to aid in the rapid development of software, they could turn out a prototype in short order. Then, after experimentation, they might go back into Sire and extend their specification. This can be done over and over again until the Sire users are satisfied. This prototyping can also be done within the different subsystems of Sire, such as the PDS subsystem, in order to make sure that the output of each stage is correct.

Reliable

It is very hard to determine if the implementation is reliable or not. At least it is hard to determine reliability in any short period of time. It seems that although Sire has been well designed, reliability can only be determined after heavy usage. The partial implementation of Sire has been used and it appears that so far it runs reliably and that the produced software is what has been expected.

6.5 Lessons Learned

1. The use of menus is a very useful concept, however, it can be detrimental. Menus are usually bothersome and a waste of time to the experienced user. An alternative to the full time menu might be some kind of toggle in which the user, or the system, can set the level of detail in which the menus are presented. This toggle concept can be expanded to all of a project. For example, all system messages could have two levels of detail controlled by this toggle.

2. An on-line help facility is a necessity. Again, this is a facility that is more likely to be used by the

beginner, but may sometimes be used by the expert.

3. One major objective of a user friendly environment should be to strive to do as much as possible in a background mode. The user should not have to wait on tasks to finish if it isn't necessary.

4. Going along with the previous lesson is the observation that the environment should take full advantage of multi-processing capabilities. This would enhance speed and user friendliness by combining several steps that had happened in serial into parallel tasks. An example of this might be that a background task is compiling the MIDAS specification while the user is using the editor to enter it, thus reducing the serial nature of the edit/compile cycle and providing for faster feedback.

5. Something very critical to the user interface is the determination as to how much should be told to the user. There are times when tasks are taking place and there is no action required from the user. The question is, what should the system be telling the user to keep user interest up without telling the user unnecessary information? The user must be informed enough to know what errors were made, but the user doesn't need to know about internal processing events that are of no help in debugging user errors.

6. A high degree of integration is hard to achieve without a high degree of concentration on that goal.

7. The use of pre-existing system tools can be more harm than help. In using these tools, the integrator must fully understand how the tool works in order to lessen the chance that the integration of the tool will have strange side effects on the system.

8. There needs to be some sort of graphical tools integrated into a software environment. The most appropriate area for these tools would be in the presentation of information to the user. For example, after the MIDAS has been correctly specified in a level of the PDS, a graphical tool might produce structure charts to depict the MIDAS structure. Also, graphical tools are very useful in the documentation that Sire must produce.

9. The environment should include a built in pretty-printer that automatically formats all source files into the format accepted at the using installation. This implies that the pretty-printer must be able to accept user input that will alter the configuration of the formatted output.

10. Related to the last lesson is the concept that most of the system tools must be configurable by the users at

the using installation. This concept must be designed in from the very beginning and should have been an objective of Sire.

11. Interaction with the system should be maximized in order to minimize batch, or serial, actions. In PDS edit/cycles, for example, the user must exit one tool, enter another, make the correction, then go back to the first tool in a repetitious manner until the problem is solved. With proper interaction, these wasted action could be reduced. However, interaction should be used only when necessary. It should not be used just to give the user a sense of usefulness.

12. The user needs to be able to set some actions running in an unattended fashic . Many of the tasks that the system performs do not need the user. Therefore, the user should be able to go away and do more productive tasks than staring at the screen. It might also be useful to consider letting the user schedule tasks for some future time and date.

13. The integration of a code control system is a necessity. In such a system, all out of date work is archived in case something happens to the working copy so that the user can easily retrieve older code and will not have to duplicate past actions. A good example of such a system is the UNIX Source Code Control System

(SCCS).

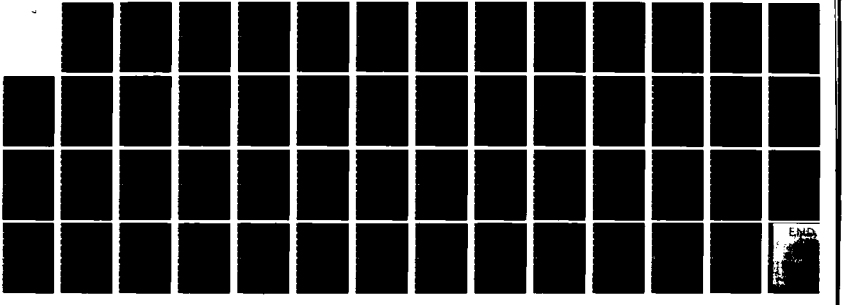
6.6 Summary

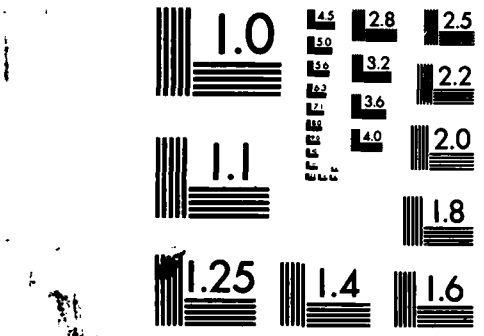
In any project there are things that are done right and things that are done wrong. It is usually easy to identify those that are done wrong as the weaknesses of the system. However, identifying the things that are done right with the strengths is often overlooked. It is not very satisfying to say that the strengths of a system are that the implementation and design turned out as expected. Because of this, the analysis of Sire takes on a decidedly negative tone. It should be stressed however, that this project has met and exceeded all the expectations that were set forth in the start of the project.

With this in mind, one must be very careful in deciding about the quality of a system when performing an analysis. In this chapter several seemingly disastrous flaws have been discussed. It should be pointed out that most of these flaws could have been worked out given more time.

AD-A138 022 SIRE: AN AUTOMATED SOFTWARE DEVELOPMENT ENVIRONMENT(U) 2/2
AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL
OF ENGINEERING D W NETTLES DEC 83 AFIT/GCS/MA/83D-5

UNCLASSIFIED F/G 9/2 NL





Chapter 7

Conclusions and Recommendations

7.1 Introduction

The purpose of this project has been to investigate automated software development environments (ASDE). Occurring somewhat in parallel with this investigation was an effort to specify, design and partially implement an ASDE, building on lessons learned from the investigation. The Sire system, as designed and implemented, has strengths and weaknesses as discussed previously. It appears that this partial design and implementation should not be a basis for continued development. This is because of the fact that the time limitations under which this project was implemented forced an implementation that could be greatly improved upon in such areas as efficiency and user interaction. Using this implementation as a basis for future projects would cause more problems than it would be worth. Instead, it should be used as a learning tool and technology demonstrator.

7.2 Conclusions About Sire

Several things seem immediately obvious when reflecting on this project. The first is that, to design and implement a useful ASDE is an extremely complicated and long-term task. It is so complicated because an ASDE not only has to deal with software engineering, efficiency, implementation of an algorithm and performance as most software projects do, it also has to deal very heavily with ergonomics, and the problems of integrating diversity.

Integrating diversity is a troublesome problem that became apparent from the very start of this project. This problem deals with having to integrate several very diverse functions into one system and having the system be easy to use and "smooth" to use. This problem is not apparent in most software projects because the majority of these projects deal with only a limited set of closely related functions. One good example of a software system that has to integrate diverse functions is the operating system. It should be easy to see that those that integrate these functions are easy to use, while those that don't handle it well are not. For example, the UNIX operating system tries to handle everything as files. As a result, UNIX presents a very easy to work with interface. This is not to say that UNIX is simple to use or

understand, for it performs the purpose it was designed for very well.

This brings up another conclusion, that of properly targeting the user level. It is important to decide what level of user will be dealing with the system and design the user interface for that level. Trying to be all things to all people will increase the complexity of the project by orders of magnitude. Therefore, it seems to be very important to not try and "design down" to the less experienced user if that user is below the targeted level.

The analysis performed in the previous chapter seems to indicate that the Sire implementation is about typical of any project that is in its infancy, as Sire is. There are problems and strengths. Many of the problems are easy (now) to attribute to the discussion above. An ASDE is a very large concept, and the complexity of the project, at times, overwhelmed the design and the implementation. Expediency took over and the design consequently suffered.

7.3 Recommendations

7.3.1 Plan to Throw One Away

It is often said in the software world that "you should plan to throw the first one away." This proverb seems to fit

Sire very well. Sire is successful in that it has been useful as a learning project and technology demonstrator. However, as a basis for future implementation, it probably would be more of a hindrance that a help. It is recommended that the design and implementation be used as a learning tool in order to see what problems can occur in this type of project and to help define requirements for another ASDE.

7.3.2 Future Projects

In making recommendations about projects to develop software environments it is probably wise to try and limit the scope to something smaller than the scope covered in this project. ASDEs, being large and complicated, are hard for one person to totally design and do justice to, not to mention attempting a partial implementation. Therefore, the author can define several projects that are natural outgrowths of Sire.

The first project would be to do an in-depth requirements analysis to add to the requirements already stated for Sire. After that analysis is completed, the high level requirements model of Sire should be adjusted to meet the new requirements. The basic flavor of the current model should not be changed because it represents a generalized attempt to define requirements for an environment that supports good software engineering methods. After re-modeling the system,

an analysis should be performed in order to determine what data base management capabilities are needed. Also, a data base schema should be designed so that all the data needs of all the sub-systems are fully met.

After the model has been adjusted and the data base provided, the functional design of Sire should be reaccomplished, taking into account the lessons learned in the current functional design. This design should emphasize the proper definition and functioning of the PARS system and the relationship of the RDS, PDS and DDS. The PARS is particularly important since it is a distributed system and it's requirements affect the design of all the other parts of Sire.

The remaining projects fall into line with the design of Sire as it stands now. The projects would include the design and implementation of the RDS, PDS, DDS, and SRS. These design should be full and specific, but the implementations should only provide the core of each system. The remainder should be provided as projects done by other students.

7.3.3 General Sire Recommendations

The first recommendation is that the use of a software specification language be retained as it proved to be very useful in this project and should be investigated in greater depth. MIDAS proved valuable because it allowed for simple

description of the program structure and provided a very valuable base of knowledge that, with the use of the computer's power, can be used for driving such tools as automatic documentation devices, automatic structure chart generators, and, obviously, translators. However, MIDAS in its present form is not complete enough for a production system. It needs to be expanded to cover advanced concepts such as tasking, generic instantiation and packaging. A comparison of the Ada programming language and MIDAS would provide the necessary areas in which MIDAS needs to be expanded. Also, it seems probable that the data section of MIDAS could be reworked to provide the user with an easier method of specifying the data blueprint.

Although no feelings or ideas about the form of parts of Sire other than the PDS have been provided, it does seem that a very interesting area of investigation could form the basis for the PARS and RDS systems. It seems that Data Flow Diagrams (DFDs) bear a strong relationship to the Critical Path Method (CPM) used in project scheduling. This would be useful because the Sire might use this relationship in the following manner. The user would enter preliminary project information into the PARS system and then proceed to the RDS system. Here the user would complete the requirements definition, including the DFD requirements model which is stored by Sire.

After the RDS is finished, the user proceeds to the PDS system and uses the DFD model to specify the preliminary design. This would then give Sire information about the relationship between the module structure of the project and the DFDs. Now, using the scheduling information, the PARS would automatically calculate a preliminary CPM schedule for the project. Every time another module is defined in either the PDS or DDS, PARS would add more detail to the CPM schedule. Now, since PARS is distributed throughout SIRE, it can monitor the progress of each module in the project, thus providing the capability to always know the status of the system and how the project stands in relation to the initial schedule.

7.4 Summary

This project winds up leaving both good and bad impressions. The good impressions have to do with the successes that the project has met with. The bad impressions deal with the things that needed to be done, but could not be done because of the lack of time. Much was learned about software environments and, especially, the complexity that large projects can bring about.

Bibliography

Abrahams, P. "Structured Programming: Considered Harmful," SIGPLAN Notices, Vol 10, No. 4. New York: Association of Computing Machinery, Inc., (April 1975), pp 13-27.

ANSI. IEEE Standard Pascal Computer Programming Language. New York: IEEE, Inc., 1983.

Barrett, William A. and John D. Couch. Compiler Construction: Theory and Practice. Chicago: Science Research Associates, 1979.

Dijkstra, Edsger W. "The Humble Programmer," Communications of the ACM, Vol 15, No. 10. New York: Association of Computing Machinery, Inc., (October 1972), pp 859-866.

DoD. Ada Reference Manual. Department of Defense, 1980.

Gane, Chris and Trish Sarson. Structured Systems Analysis. New York: Yourdon Press inc., 1978.

Gutz, Steve, Michael J. Spier and Anthony I. Wasserman. The ergonomics of Software Engineering - Description of the Problem Space" Software Engineering Environments. edited by Hunke Horst. New York: North Holland Publ., 1981.

Hammer, Michael and Gregory Ruth. Research Directions in Software Engineering. edited by Peter Wegner. Cambridge, Mass: MIT Press, 1980.

Infotech International Limited. Software Engineering Techniques, Vols 1. Maidenhead, Berkshire, England: Infotech International Limited, 1977.

Kernighan, Brian W. and Dennis M. Ritchie. The C Programming Language. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1978.

Lanergan, Robert G. and Dennis K. Dugan. "Software Engineering With Reusable Designs and Code" IEEE COMPCON FALL. (1981).

Myers, Glenford J. Software Reliability. New York: John Wiley and Sons, Inc., 1976.

Peters, Lawrence J. Software Design: Methods and Techniques. New York: Yourdon Press Inc., 1981.

Wasserman, Anthony I and Peter Freeman. Interface Workshop on Software Engineering Education. New York: Springer-Verlag Inc., 1976.

Wasserman, Anthony I. "On the Meaning of Discipline in Software Design and Development" Software Engineering Techniques, Vol. 2. Maidenhead, Berkshire, England: Infotech International Limited, (1977).

Wasserman, Anthony I. Tutorial: Software Development Environments. edited by Anthony I. Wasserman. New York: IEEE Computer Society Press, 1981.

Weinberg, Victor. Structured Analysis. New York : Yourdon Press, 1978.

Yourdon, Edward Techniques of Program Structure and Design. Englewood Cliffs, New Jersey: Prentice-Hall, inc., 1975.

Yourdon, Edward and Larry L Constantine. Structured Design, 2nd Ed. New York, New York: Yourdon Press, 1978.

Zelkowitz, Martin V. Principles of Software Engineering and Design. Englewood Cliffs, New Jersey: Prentice-Hall, inc., 1979.

Zelkowitz, Martin V. "Rapid Prototyping Woprkshop: Overview" Software Engineering Notes, Vol. 7, No. 5. New York: Association of Computing Machinery, Inc., (December, 1982), p2.

Appendix A

Glossary of Terms

Backus Naur Form (BNF)
A means of formally describing the syntax of a language.

BNF Backus Naur Form.

Code Generator
Tools that assist in the production by automatically translating from one form to another target form.

Detailed Design
A specific statement of the methods that will be used in the implementation of the software.

Detailed Design Document
A formal document describing the detailed design.

DFD Data Flow Diagrams. A graphical method for representing the flow of information in a system. Often used to state the requirements of a system.

Implementation
The stage of the project in which the design is coded in some implementation language.

Interface Checkers
Tools that automatically check and verify the interface between different modules.

Linkers Tools that merge and join separate modules, usually object modules.

Maintenance/Operation
The stage of the project when the software is being used and changes are being made to correct errors or make changes.

Preliminary Design
: A description, usually graphical, of the structure of the system.

Preliminary Design Document
: A formal description of the preliminary design.

Software Engineering
: A discipline that tries to develop proper engineering methods by which quality software may be produced.

Source Code
: Human readable representation of the software.

Structure Charts
: A graphical method representing the design of software systems by showing the hierarchical structure and the flow of information and control.

Structured Code
: Source code that is created using rigid, formal methods in the hopes of constructing quality software. Usually having to do with program layout and using single entrance points and single exits.

Syntax-Directed Editors
: Interactive automated tools that check the syntax of the source code as it is entered in the computer by comparing the code with the syntax description for the language.

Test Plans
: A formal statement of the methods, strategies and test cases that will be used to validate the software system.

Text Editors
: Interactive tools that allow the input and modification of text into computer files.

Textual Requirements
: A formal statement of the requirements of the system in english, or an english-like language.

Tool Kit Approach
: An approach to developing software that emphasizes the use of many small software components integrated into a single system.

Appendix B

MIDAS Language Description

(Modular Interface Definition And Specification)

B.1 Definitional Conventions

The metalanguage used in this description of the MIDAS syntax is based on Backus-Naur Form (BNF). It has been modified to permit greater convenience of description. The meanings of the various metasymbols are given below.

Metalanguage Symbols

Metasymbol	Meaning
::=	shall be defined to be
|	alternatively
.	end of definition
[x]	0 or 1 instances of x
{x}	0 or more instances of x
(x|y)	grouping: either x or y
xyz	terminal symbol xyz
)xyz)	also terminal symbol xyz
<meta-identifier>	nonterminal symbol

B.2 Lexical Tokens

Special Symbols

```
     <letter> ::= a | b | c | c | d | e | f | g |  h | i | j |
k | l | m | n | o | p | q | r | s |  t | u | v | w | x | y | z
| A | B | C | D | E | F | G | H | I  | J | K | L | M | N | O |
P | Q | R | S | T | U | V | W | X | Y | Z | ' '.

     <digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9.

     <symbol> ::= + | - | * | = | < | > | [ | ] | . | , | : |
; | | | ( | ) | => | .. | ) | '.

     <special_symbol> ::= <symbol> | <word_symbol>.

     <word_symbol>  ::= SYSTEM | SUB-SYSTEM | SPECIFICATION |
IS |

     <hierarchy_word_symbol>     |     <mid_word_symbol>     |
<data_word_symbol>.
```

It is important to note here that <word_symbol> varies depending on which section of MIDAS is currently active. The implication here is that there will be a different set of reserved words in the hierarchy section, mid section and the data section. This will be discussed in greater depth at a later time.

It is important to note that the syntax of MIDAS is, at times, very restrictive and demanding. The reason for this is that most of the MIDAS description is not intended to be

filled out by the user, instead it will be filled out by the system.

<hierarchy_word_symbol> ::= STAND | ALONE | separate | LEVEL | NOT | IS | DEFINED | AS | SYSTEM | HIERARCHY | DESCRIPTION | COMPLETELY | DESCRIBED | SUB.

<mid_word_symbol> ::= PURPOSE | ALIAS | AS | BLOCK | BY | CALLS | CASE | CALLED | COMMENT | COMPLETELY | CREATED | CREATOR | DEFINED | DESCRIBED | DESCRIPTION | ELSEWHERE | EMPTY | END | FAN | FAN | IN | INTERFACE | IS | MODULE | NAME | NOT | OUT | USER |.

<data_word_symbol> ::= CASE | ARRAY | AS | BLUEPRINT | BOOLEAN | CHARACTER | CONSTANT | COMPLETELY | DATA | DELTA | DESCRIBED | END | FILE | INTEGER | LENGTH | NOT | OF | OTHERWISE | NULL | POINTER | RANGE | REAL | RECORD | STRING | TO | TYPE | WHEN | WITH .

Identifiers

Identifiers may be of any length. All characters of an identifier are significant in distinguishing between identifiers. Any <word_symbol> is required to be in all capital letters. Therefore, an identifier with the same spelling but with different case is separate distinguishable from the <word_symbol>. In other words, case is significant. The reason for this is for flexibliity. Sometimes an identifier may be used in one place and the user may wish to refer to it later in the comment section. This gives the flexibility necessary to accomplish this.

EXAMPLE:

	SPECIFICATION
is not the same as	Specification
or	SPECification
but	Specification
is the same as	SPECification

Numbers

An unsigned integer shall denote in decimal notation a value of integer type. An unsigned real shall denote in decimal notation a value or real type (float type).

```
<signed_number> ::= <signed_integer> | <signed_real>.

<signed_real> ::= [<sign>] <unsigned_real>.

<signed_integer> ::= [<sign>] <unsigned_integer>.

<sign> ::= + | -.

<unsigned_real> ::= <unsigned_integer> '.' <fractional_part>.

<unsigned_integer> ::= <digit_sequence>.

<digit_sequence> ::= <digit> {<digit>}.

<fractional_part> ::= <unsigned_integer>.
```

Character Strings

A character string containing a single string element shall denote a value of character type. A character string containing more than one string element shall denote a value of string type.

```
<character_string> ::= '"'<string_element>'"' |
        '"'<string_element>{<string_element>'"'.

<string_element> ::= <letter> | <space>.
```

EXAMPLES:

```
Character types  'A'   ' '   ';' ')'
String types     "A"   " " "this is a legal string type" "\"".
```

Note that the \ character is used as an escape to allow the placement of the " character (quote character) into a string.

B.3 Syntax Description

The following is BNF description of the MIDAS specification language.

```
<midas> ::= MIDAS <system_identifier> SPECIFICATION IS
        <hierarchy_description>
        <data_blueprint>
        <module_interface_description>.
```

This is the MIDAS header which describes the relationship between the three principle parts of MIDAS.

```
<system_identifier> ::= <identifier>.

<hierarchy_description> ::= HIERARCHY DESCRIPTION IS
        [NOT] COMPLETELY DESCRIBED [AS <hierarchy>]';'.

<hierarchy> ::= "(" <system_level> <system_identifier>
        IS <level_description> ")" {<level_breakdown>}

<system_level> ::= SYSTEM | SUB-SYSTEM.
```

The level of the system is used in describing whether or not the part being described is the system or just part of the system.

```
<level_description> ::= <module_name> {<module_name>}.

<module_name> ::= <identifier>.

<level_breakdown> ::= '(' <module_name> IS <level_option> ')'.
```

The <level_breakdown> describes the level's makeup.

```
<level_option> ::= (SEPARATE [STAND ALONE] LEVEL)
        | <level_description>.
```

If a level is specified as separate then it means that the level will be completely described in another hierarchy description. The stand alone option indicates whether or not the separate level will be a stand alone, operational system, or whether it is a collection of modules.

This completes the syntax description of the hierarchy part of MIDAS. The next description will be for the mid part of MIDAS.

```
<module_interface_description> ::= MODULE INTERFACE DESCRIPTION
        IS [NOT] COMPLETELY DESCRIBED AS {<repeated_description>}.

<repeated_description> ::= [<system_level>] MODULE <identifier>
        IS [NOT] COMPLETELY DEFINED <where_defined>";".

<where_defined> ::= (AS <description> END MODULE
        <identifier>) | ELSEWHERE.
```

Either a module is defined here, in this mid description, or it is described in another mid description.

```
<description> ::= <interface> <comment_block>.

<interface> ::= (INTERFACE IS <interface_description>";") | EMPTY';'

<interface_description> ::= {<identifier> {, <identifier>} : [<mode>
        <type_identifier>';'}.

<mode> ::= IN | OUT | IN OUT.

<comment_block> ::=
     /* MODULE COMMENT BLOCK **********************
        MODULE NAME => <identifier>
        MODULE ALIAS => (<identifier> | none)
        MODULE CREATED => <date>
        MODULE CREATOR => <creator_identifier>
        MODULE FANOUT => <number>
        MODULE FANIN => <number>
        MODULE CALLED BY => [<identifier> {, <identifier>}]
        MODULE CALLS => [<identifier> {, <identifier>}]
        MODULE PURPOSE => '"' <module_descriptor> '"'
     ***********************************************/

<date> ::=

<creator_identifier> ::= <identifier>.
```

```
<module_descriptor> ::= {(<letters> | <word_symbol> | <special_symbol
        | <number>)}
```

This completes the syntax description of the mid part of MIDAS. The next description will be for the data part of MIDAS.

```
<data_blueprint> ::= DATA BLUEPRINT IS [NOT] COMPLETELY DESCRIBED
        AS <data_completion>.

<data_completion> ::= [<constant_section {constant_section}]
        type_description {type_description}.

<constant_section> ::= CONSTANT <identifier> IS <constants> ';'.

<constants> ::= <signed_number>
        | '"' <character_string> '"'
        | ''' <character> '''.

<type_section> ::= TYPE <identifier> IS <type_denoter> ';'.

<type_denoter> ::= <type_name> | <new_type>.

<new_type> ::= <pointer_type>
        | <enumeration_type>
        | <array_type>
        | <file_type>
        | <record_type>
        | <subrange_type>.

<type_name> ::= <identifier>
        | REAL
        | INTEGER
        | BOOLEAN
        | CHARACTER
        | <string_type>.

<pointer_type> ::= POINTER TO <type_name>.

<enumeration_type> ::= '(' identifier_list ')'
        | '(' character_list ')'.

<identifier_list> ::= <identifier> {, <identifier> }.

<character_list> ::= '(' <letter>|<digit>|<symbol>
        {, <letter>|<digit>|<symbol> <letter>|<digit>|<symbol>} ')'.

<array_type> ::= ARRAY '[' index_selector ']' OF <type_name>.

<index_selector> ::= <ordinal_type> {, <ordinal_type> }.
```

```
<ordinal_type> ::= <type_name>
                 | <enumeration_type>
                 | <integer_subrange>.

<file_type> ::= FILE OF <type_name>.

<subrange_type> ::= <integer_subrange> | <real_subrange>.

<integer_subrange> ::= INTEGER RANGE <signed_number> ".."
        <signed_number>.

<real_subrange> ::= REAL RANGE <signed_real> ".." <signed_real>
        [ WITH DELTA <unsigned_real> ].

<string_type> ::= STRING [ LENGTH <unsigned_integer> ".."
        <unsigned_integer> ].

<record_type> ::= RECORD <component_list> END RECORD.

<component_list> ::= [ <fixed_part> ] [ <variant_part ].

<fixed_part> ::= <fixed_component> {<fixed_component>} | NULL.

<fixed_component> ::= <identifier_list> ':' <type_denoter> ';'.

<variant_part> ::= CASE <type_name> IS <when_part> { <when_part }
        [<default>] END CASE ';'.

<when_part> ::= WHEN <constant_list> "=>" <component_list>.

<constant_list> ::= <repeated_constant_part> | <identifier_list>.

<repeated_constant_part> ::= <constants> {, <constants> }.

<default> ::= OTHERWISE "=>" <component_list>.

<string> ::= <identifier>.

<identifier> ::= <letter> { '_' | <letter> | <digit> }.
```

B.4 Examples

```
***** SAMPLE HIERARCHY DESCRIPTION ****

HIERARCHY DESCRIPTION IS COMPLETELY DESCRIBED AS
```

```
( SYSTEM sire IS pars rds pds dds srs utility_tools report_error )

  ( pds IS menu editor hierarchy mid data utility_tools report_erro
  ( hierarchy IS report_error compile_hierar )
  ( compile_hierar IS resolve_module install lookup attach new_hier_
    gen_mid_desc report_error )
  ( gen_mid_desc IS enter_module check_swap )
  ( enter_module IS load_swap )
  ( mid IS compile_mid report_error )
  ( compile_mid IS resolve_module make_data_desc report_error )
  ( make_data_desc IS check_known )
  ( data IS write_table make_release_d load_table enter check_for_na
    perform_tab_se enumeration_en check_table_fu report_error releas
  ( load_table IS enter )
  ( enter IS resolve_confli )
  ( enumeration_en IS enter )
  ( utility_tools IS menu list_projects list_project_1 list_lib_modu
    list_undef_mod remove_proj rem_proj_level rem_mods query_errno
   query_language set_language );


***** SAMPLE MODULE INTERFACE DESCRIPTION *****


MODULE INTERFACE DESCRIPTION IS COMPLETELY DESCRIBED AS

MODULE compile_hierar IS COMPLETELY DEFINED AS

    INTERFACE IS
 one, two, three : IN STRING,
 four : IN OUT REAL,
 five : IN new_pointer;

  /* MODULE COMMENT BLOCK ***********************
     MODULE NAME => compile_hierar
     MODULE ALIAS => none
     MODULE CREATED => Thu Nov 10 12:56:54 1983
     MODULE CREATOR => USER = nettles
     MODULE FAN-OUT => 7
     MODULE FAN-IN => 1
     MODULE CALLED BY =>  hierarchy
     MODULE CALLS =>  resolve_module, install, lookup, attach,
       new_hier_desc, gen_mid_desc, report_error
     MODULE PURPOSE => "The purpose of this module is
      to act as a handout for the briefing on SIRE."
  ***********************************************/

END MODULE compile_hierar;

MODULE list_project_1 IS COMPLETELY DEFINED AS

    INTERFACE IS
 EMPTY;
```

```
   /* MODULE COMMENT BLOCK **********************
      MODULE NAME => list_project_1
      MODULE ALIAS => none
      MODULE CREATED => Thu Nov 10 12:56:55 1983
      MODULE CREATOR => USER = nettles
      MODULE FAN-OUT => 0
      MODULE FAN-IN => 1
      MODULE CALLED BY =>  utility_tools
      MODULE CALLS =>  none
      MODULE PURPOSE => "The purpose of this module is"
    ***********************************************/

END MODULE list_project_1;

***** SAMPLE DATA DEFINITION SECTION *****

DATA BLUEPRINT IS COMPLETELY DESCRIBED AS

CONSTANT string_example IS "HELLO there my name is sire";
CONSTANT int_example IS 3;
CONSTANT real_example is 3.0;

TYPE new_pointer IS DEFINED AS POINTER TO INTEGER;
TYPE argument_pointer IS DEFINED AS POINTER TO STRING;
TYPE argument_vector IS DEFINED AS ARRAY [] OF argument_pointer;
```

Appendix C

System Design

C.1 Requirements Model

Data Node List

- 1 SYSTEM CONCEPT
  - -- 1.0 SYNTHESIZE SOFTWARE
  - -- 1.1 DEFINE REQUIREMENTS
  - -- 1.2 DEVELOP PRELIMINARY SOFTWARE
  - -- 1.4 DEVELOP DETAILED SOFTWARE
  - -- 1.5 RELEASE SOFTWARE

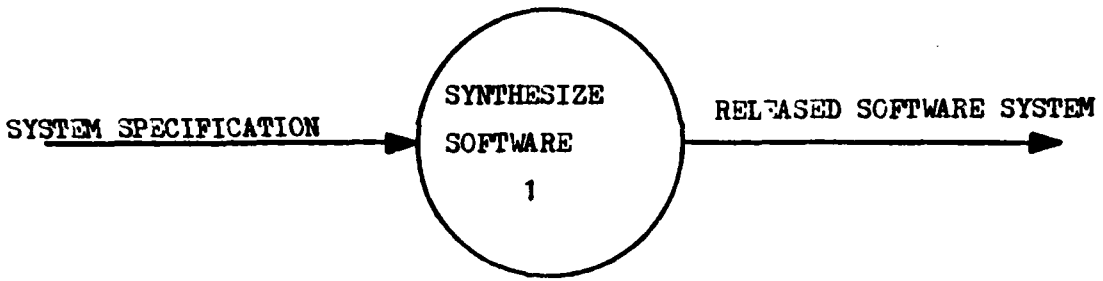


Figure 15. System Concept

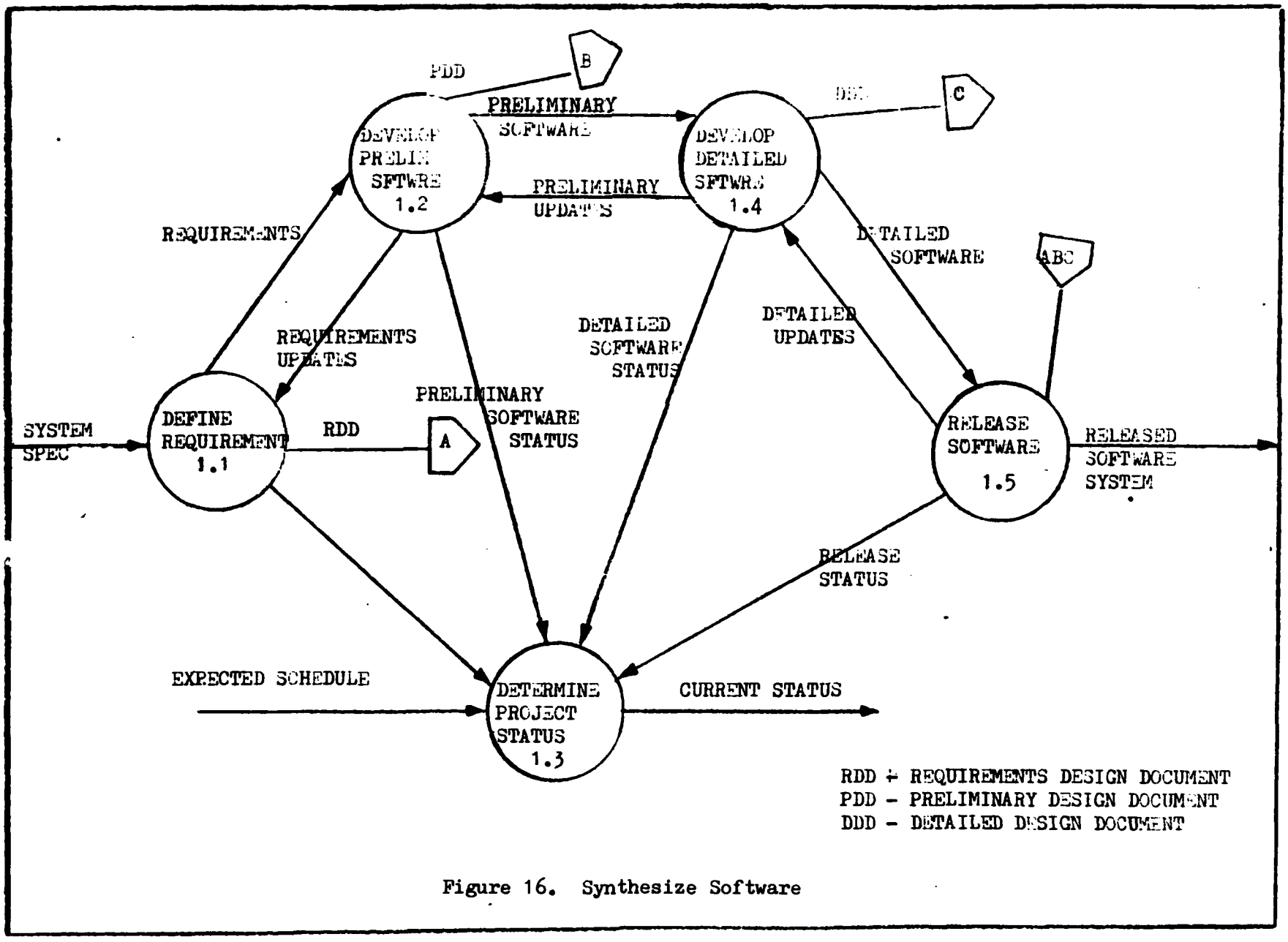


Figure 16. Synthesize Software

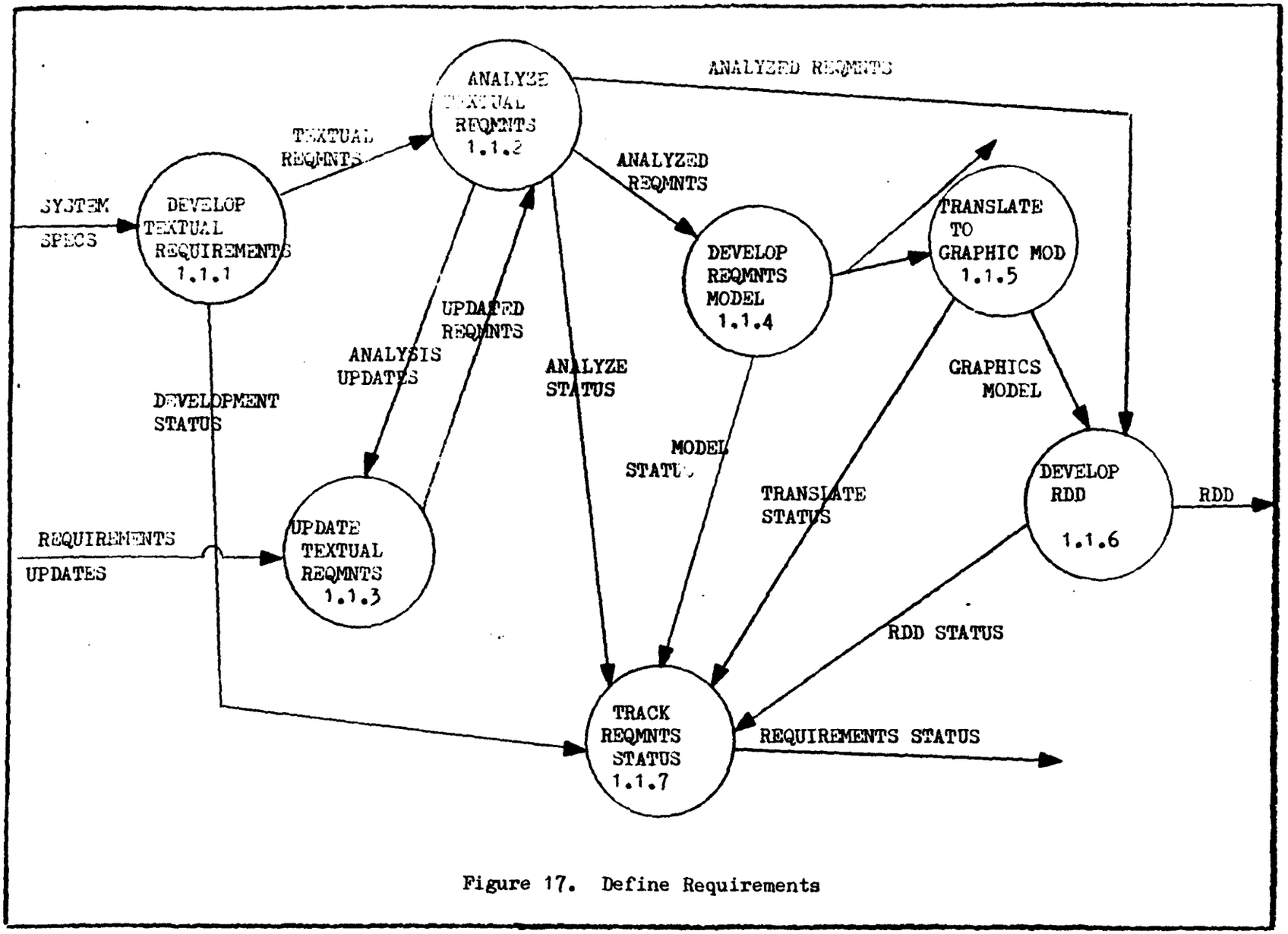


Figure 17. Define Requirements

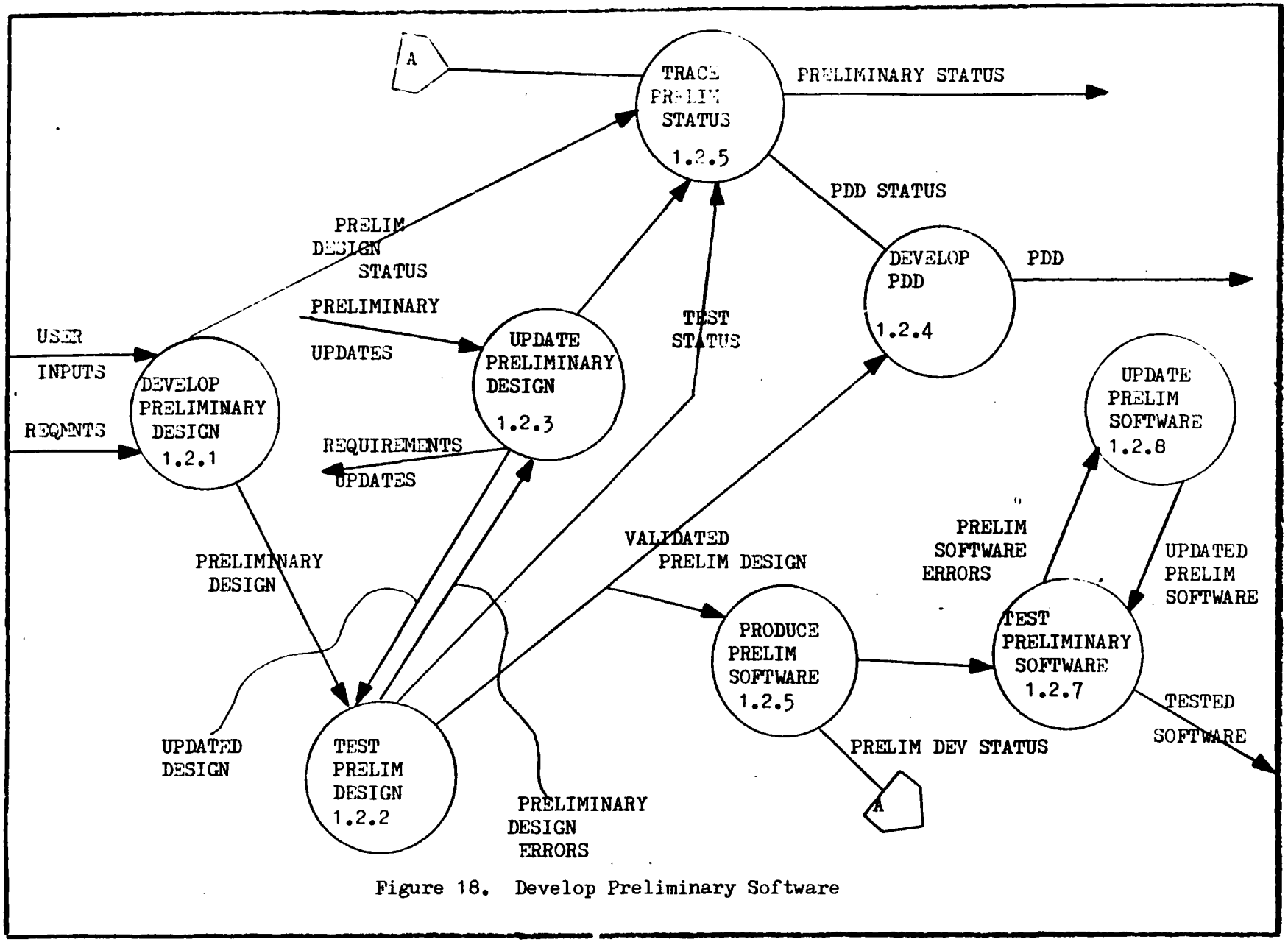


Figure 18. Develop Preliminary Software

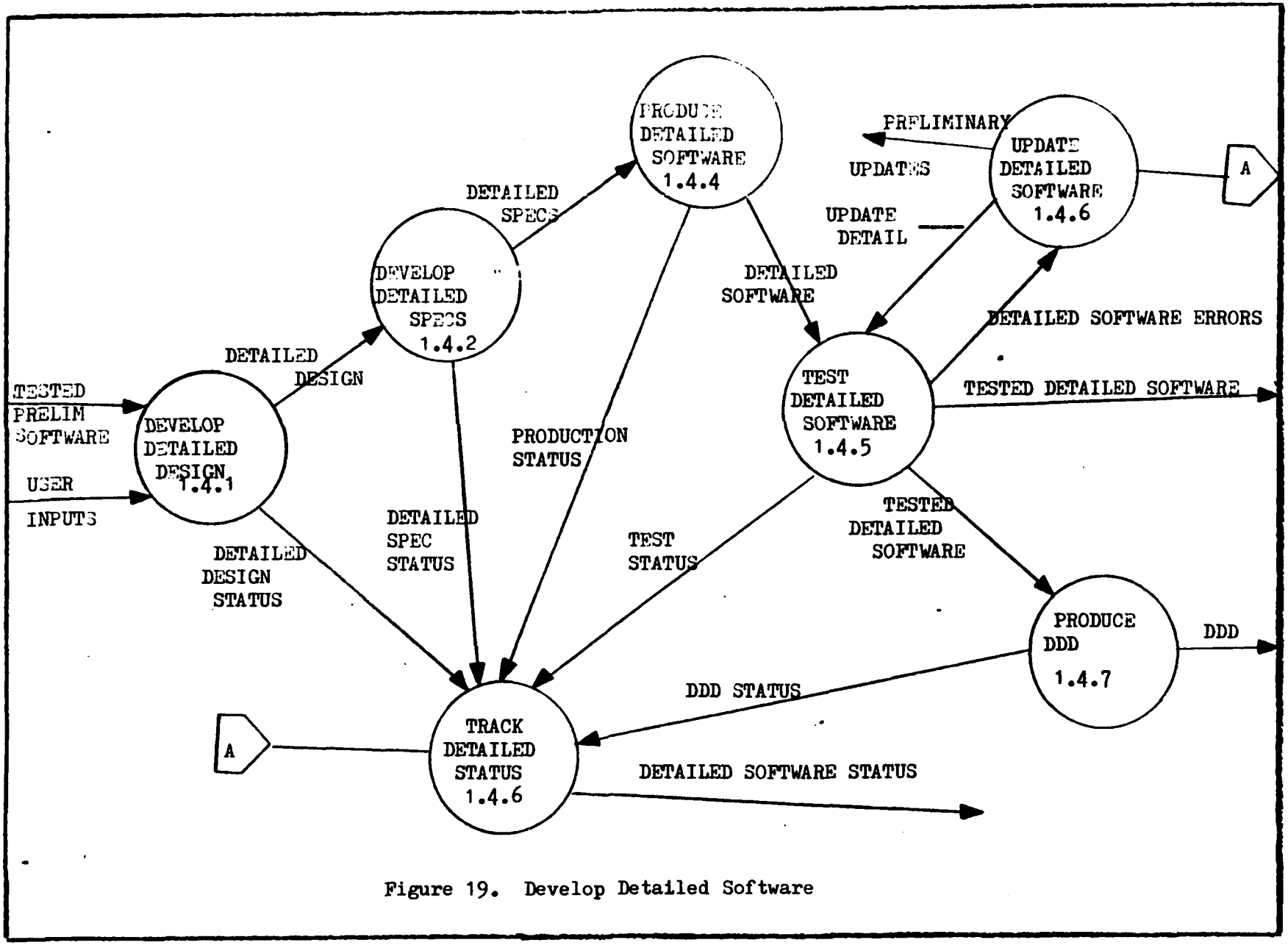


Figure 19. Develop Detailed Software

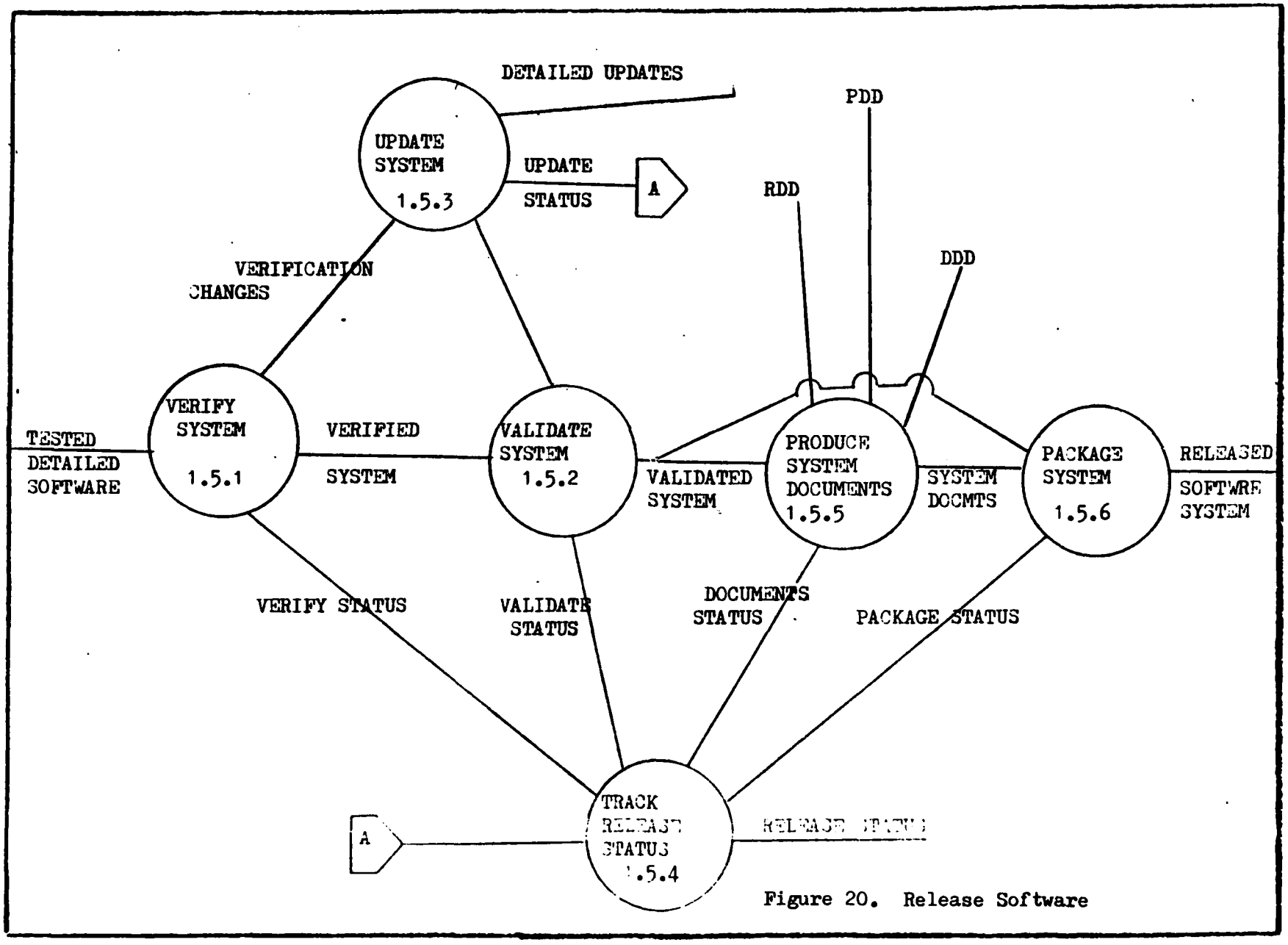


Figure 20. Release Software

C.2 Structure Charts

Structure List

- 1.0 EXEC.
  - -- 1.2 MENU.
  - -- 1.5 PDS.
    - --- 1.5.2 HIERARCHY.
      - ---- 1.5.2.2 COMPILE HIERARCHY.
    - --- 1.5.3 MID.
      - ---- 1.5.3.2 COMPILE MID.
    - --- 1.5.4 DATA.
      - ---- 1.5.4.2 COMPILE DATA.
  - -- 1.8 UTILITY TOOLS.

Data Item List

1. PROJECT NAME
   THE NAME OF THE CURRENTLY SPECIFIED PROJECT

2. LEVEL NAME THE NAME OF THE CURRENTLY SPECIFIED LEVEL

3. MENU CHOICES
   THE ACTION CHOICES THAT THE USER HAS

4. MENU CHOICE
   THE MENU CHOICE CHOSEN BY THE USER

5. VALID CHOICE
   FLAG INDICATING THAT THE CHOICE WAS VALID

6. ERROR STATUS
   THE ERROR STATUS OF THE SYSTEM

7. CURRENTLY ACTIVE SUBSYSTEM
   THE SUBSYSTEM (PARS, RDS,PDS,DDS,SRS) THAT IS CURRENTLY ACTIVE

8. SWAP LOCATION
   THE LOCATION OF THE SWAP FILE, PROJECT DEPENDANT

9. HIERARCHY LOCATION
   THE LOCATION OF THE HIERARCHY DESCRIPTION

10. HIERARCHY TREE
    THE DATA STRUCTURE THAT THE HIERARCHY COMPILER BUILDS IN MEMORY

11. SWAP FILE THE SWAP FILE

12. MID LOCATION
    THE LOCATION OF THE MID DESCRIPTION

13. MID TREE THE TREE BUILT IN MEMORY BY THE MID COMPILER

14. LEVEL TABLE
    THE TABLE OF ALL THE LEVELS OF THE PROJECT

15. RELEASE STATUS
    THE STATUS THAT DETERMINES WHETHER THE LEVEL

CAN GENERATE ANY CODE AT THE CURRENT TIME

16. DATA DESCRIPTION
THE DESCRIPTION FILE FOR THE DATA BLUEPRINT

17. DATA LIST THE DATA LISTING FILE

18. IMPLEMENTATION LANGUAGE
THE LANGUAGE INTO WHICH MIDAS IS TRANSLATED

19. TOOL LIST LIST OF ALL TOOLS

20. TOOL CHOICE
THE TOOL CHOOSEN

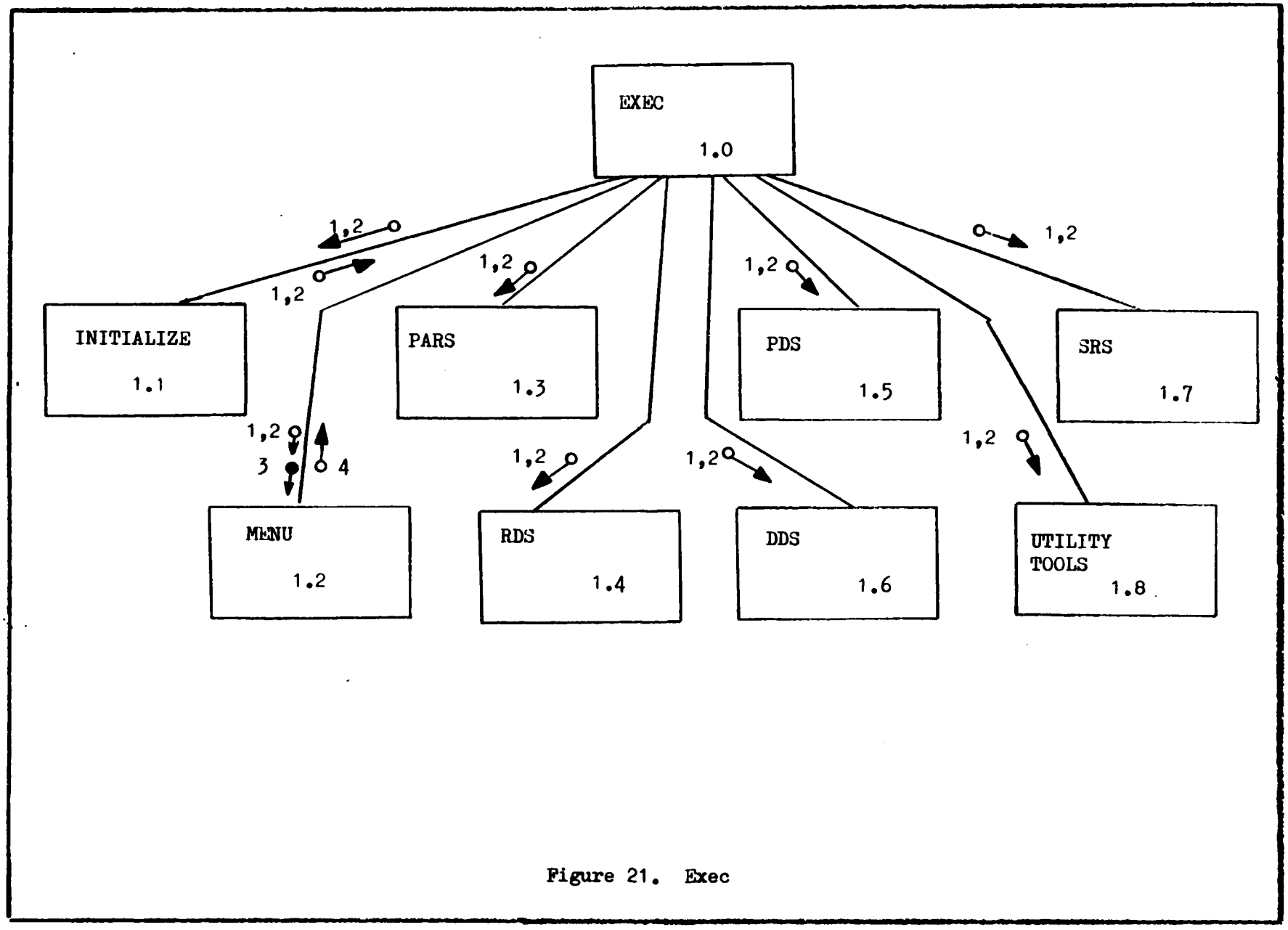


Figure 21. Exec

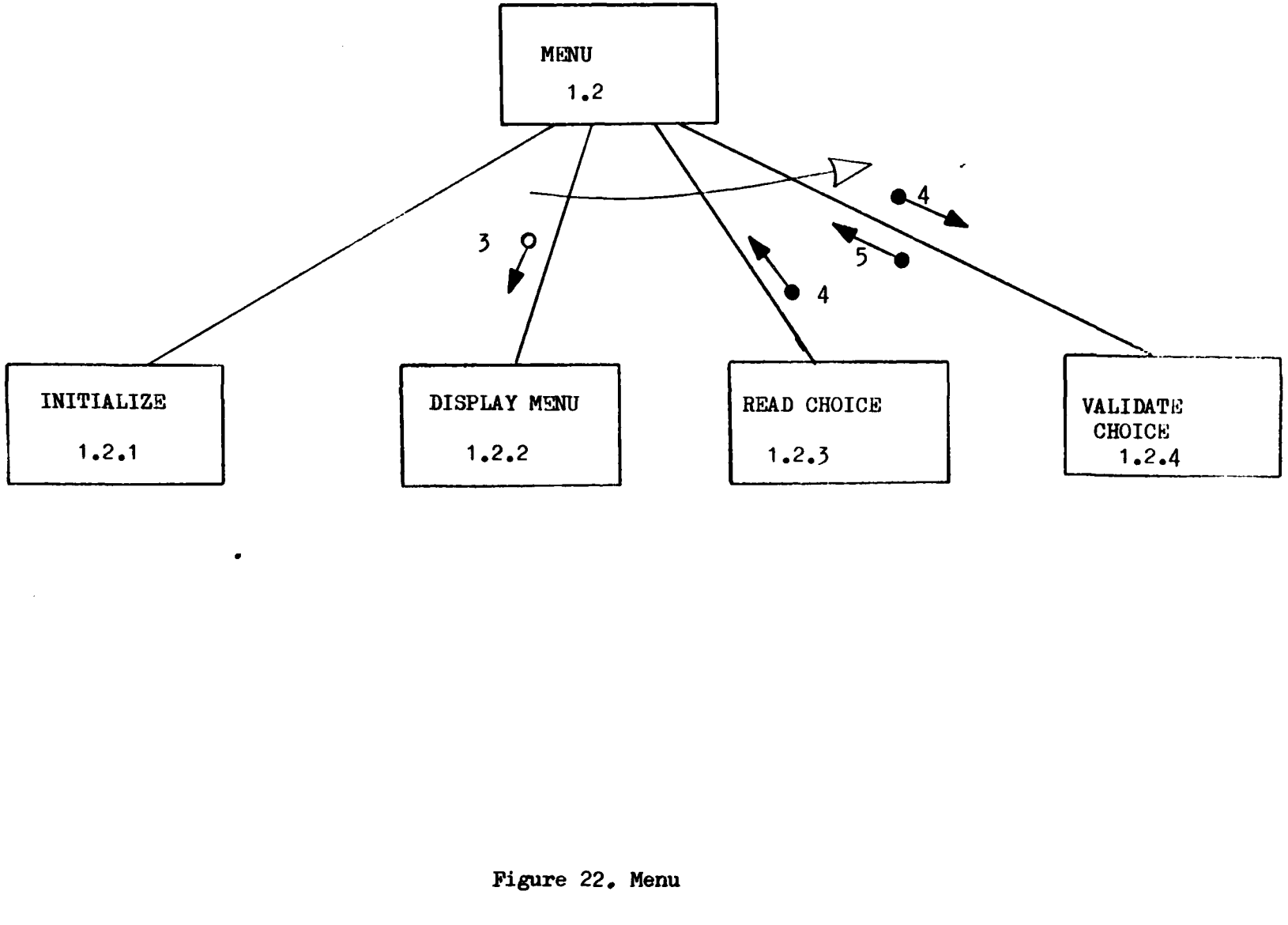


Figure 22. Menu

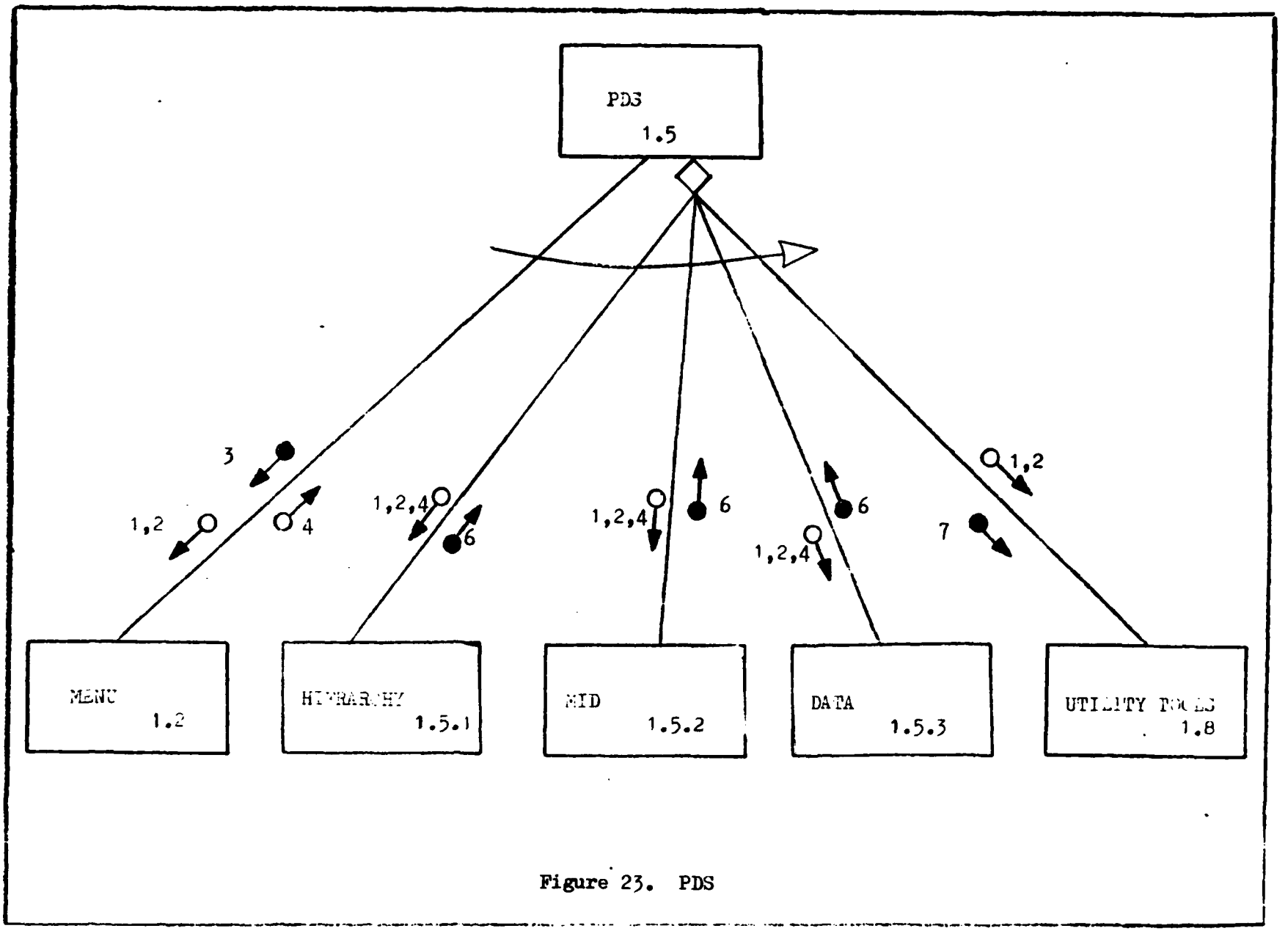


Figure 23. PDS

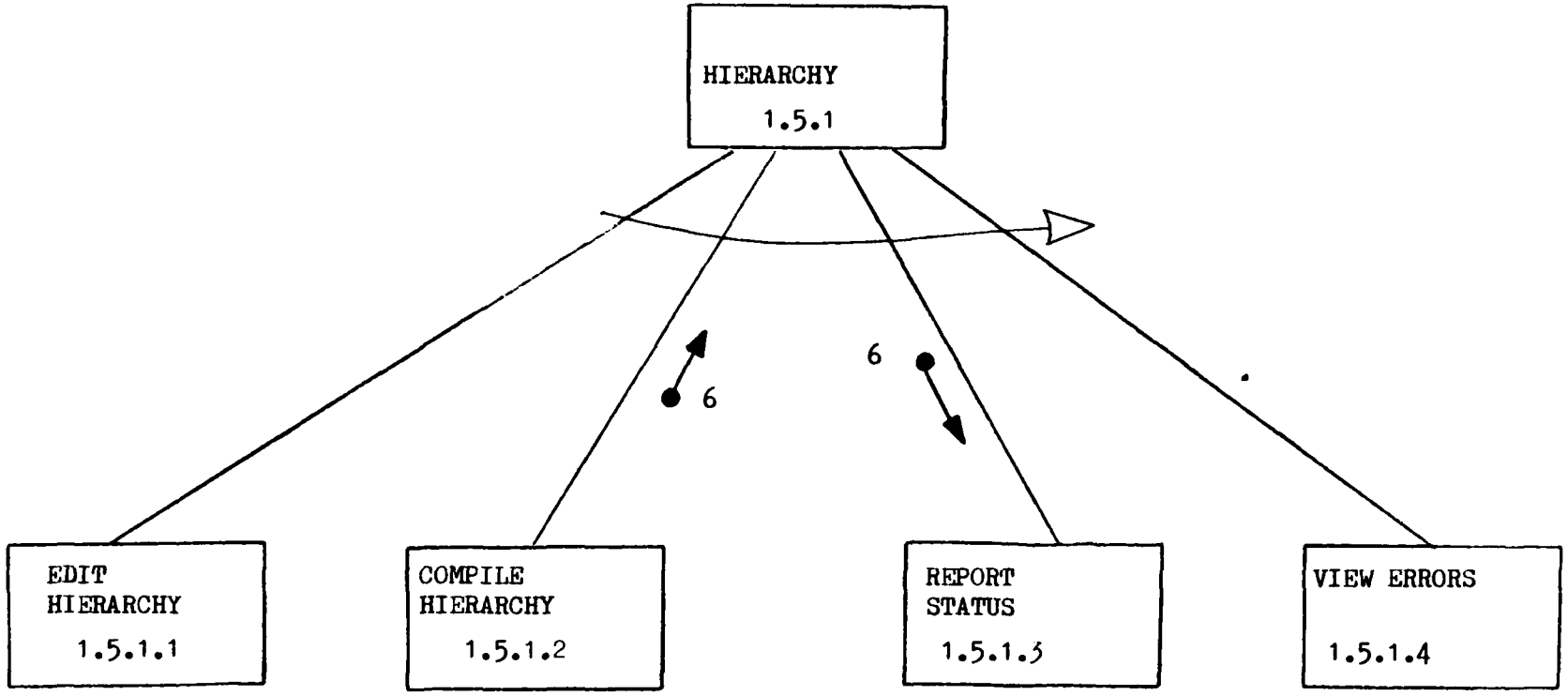


Figure 24. Hierarchy

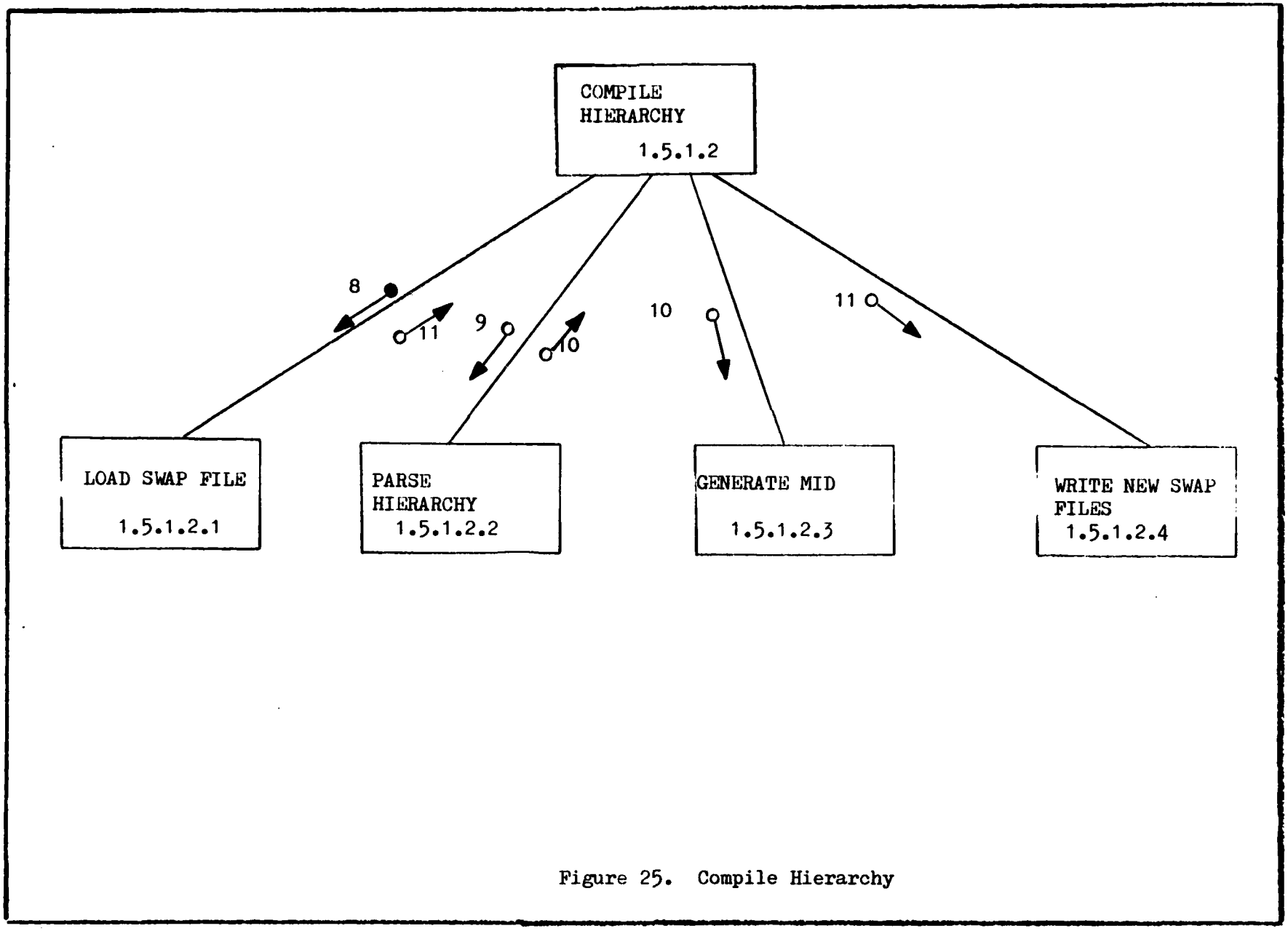


Figure 25. Compile Hierarchy

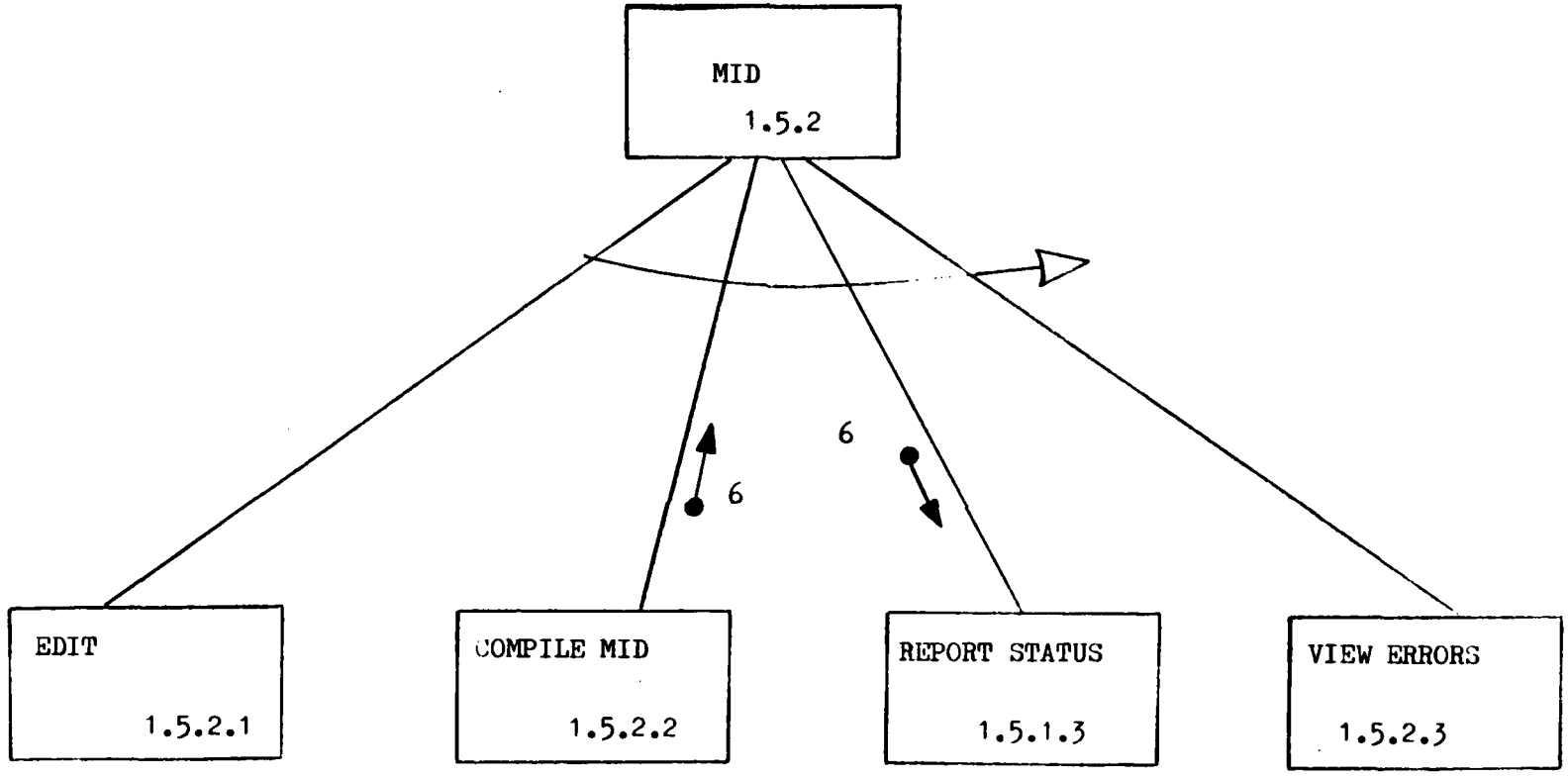


Figure 26. MID

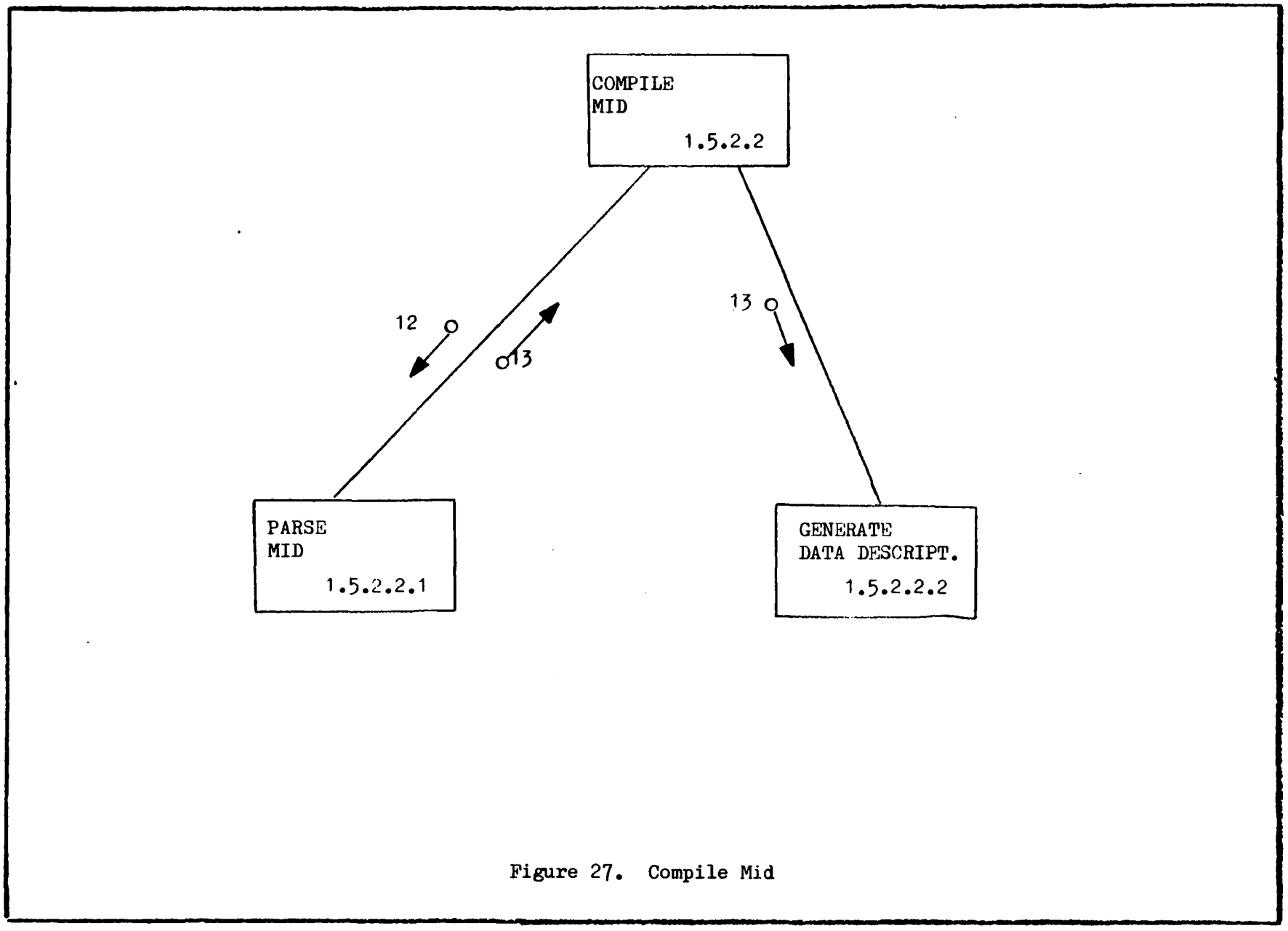


Figure 27. Compile Mid

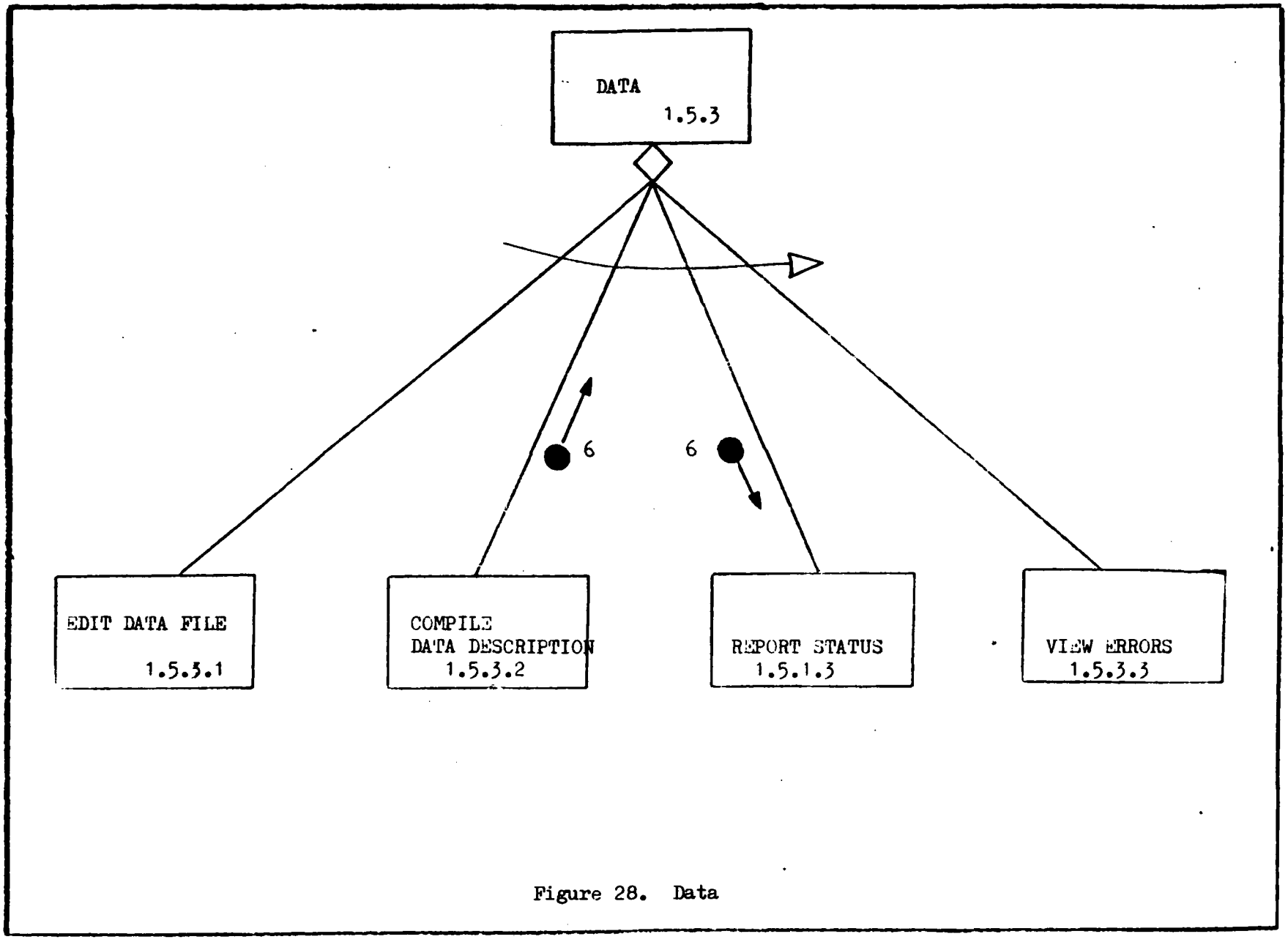


Figure 28. Data

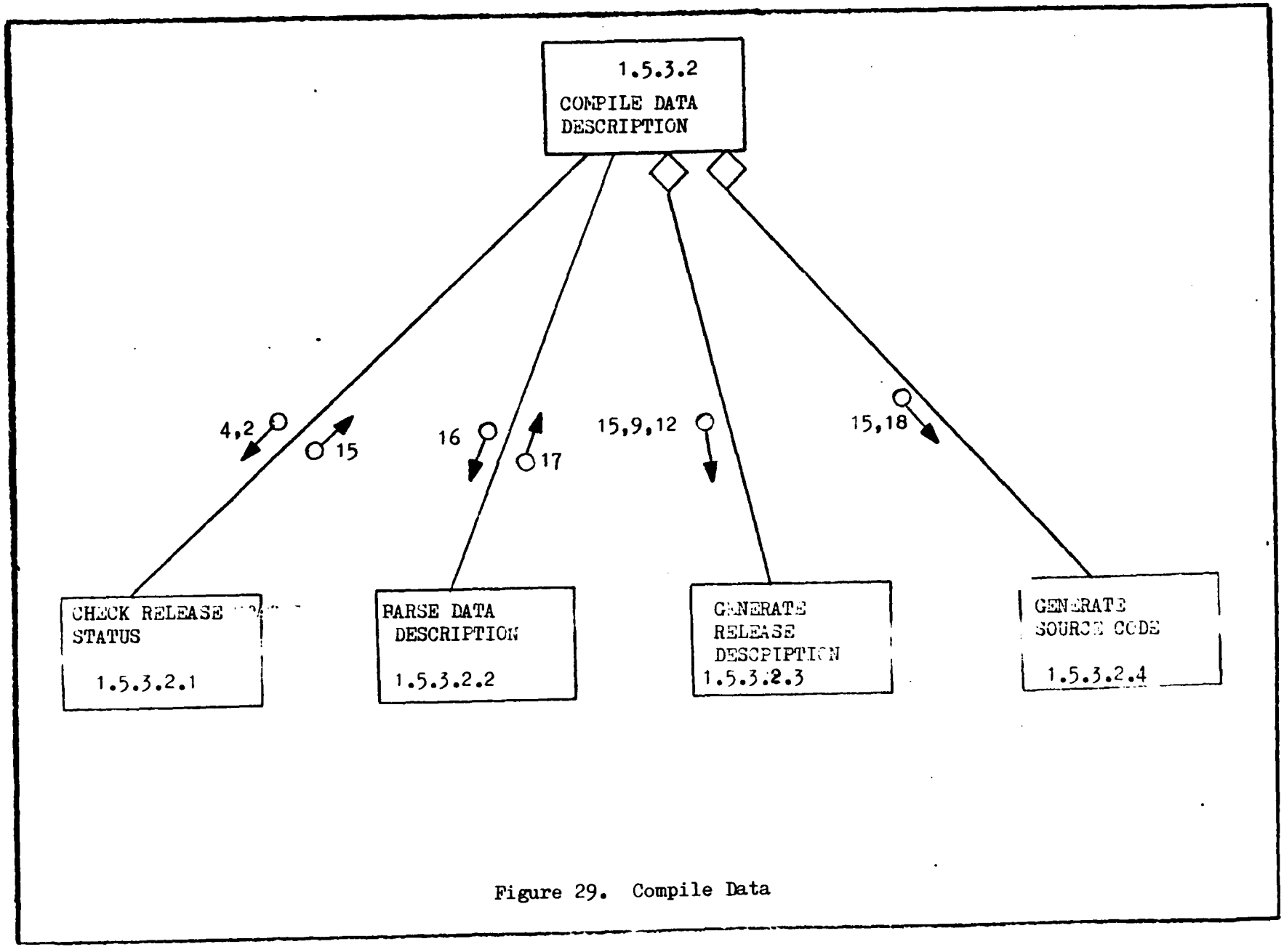


Figure 29. Compile Data

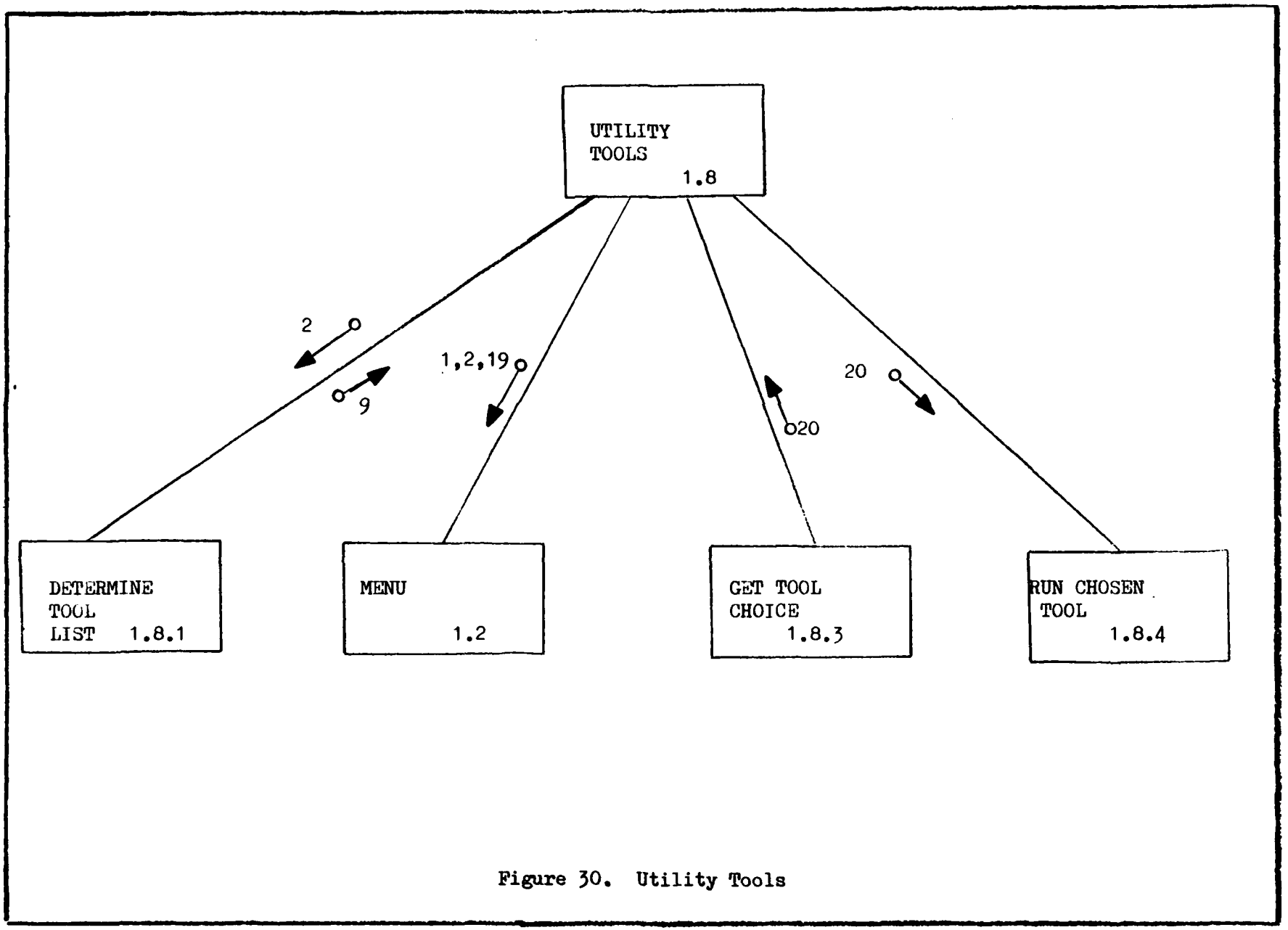


Figure 30. Utility Tools

Appendix D

Sire User's Manual

D.1 General Information

Sire is a menu driven software environment. Because of this, it is easy to use the system without having had any experience with it. Therefore, this user's manual will not be a detailed explanation of the use of this system, rather, it will be geared more toward a tutorial type of manual. Several assumptions are made about the user's knowledge at this point. The first is that the user is familiar with the MIDAS language. Secondly, it is assumed that the user is familiar with the VI editor currently being used in Sire. Finally, and least important, is the assumption that the use have some small working knowledge of UNIX.

D.2 Invoking Sire

Sire may be invoked in one of two ways. The first is to simply issue the UNIX command - sire <project_name> <user_name>. If Sire recognizes <project_name> as being a valid project, it will start up with the currently defined level being the default, main. If <project_name> is not recognized then sire will assume ask whether a new project is being started or whether a mistake has been made. If a new project is specified then action will be taken to initialize a new project, otherwise Sire will terminate.

The second method used to start Sire is basically the same, however Sire allows the specification of the project level at this time. If the project level is invalid then Sire will default to the main level. In any case, it is necessary to enter the name of the user as the last argument on the command line.

Examples of Sire invocation:

```
sire nettles      -- the basic method

sire project1 nettles     -- specifying a project name

sire project1 level1 nettles  -- specifying a project and level
```

D.3 Top Level Operation

After Sire has been invoked, the user will be presented with a menu of choices. At the top of the menu will be a message specifying the currently active project and level. The choices correspond to the different sections of Sire, with the exception of two. These two are the utility_tools and the level name change choice. The utility_tools are all easy to understand from their menu name, therefore it is easier to use them than to read about them. The name change choice lets the user change the level currently active (obviously).

All the user has to do at this level is to choose which section of Sire is needed. This will vary with the stage that the project is in, but it should proceed from the first section, pars, to the last, srs.

D.4 Operation of PARS

TBD.

D.5 Operation of RDS

TBD.

D.6 Operation of PDS

Starting PDS

After having specified the PDS menu choice in the top level of Sire, the next thing that will happen will be the appearance of the PDS menu. The operation of the PDS, like all parts of Sire, is driven completely by the menu. The next section will describe the menu and how the PDS works. The description is only the recommended course of actions. After familiarity has been gained, the user may want to vary the actions in any manner suitable to the task.

The Menu

The basic parts of the menu are the edit/compile cycles and the tools. The edit/compile cycles are responsible for building up the complete description needed by PDS in order to function correctly. The first cycle is the "hierarchy" cycle. In this cycle the user will start the editor and enter the hierarchy description that is needed. After entering the description and leaving the editor, the next action will be to compile that description. If any error messages occur, the user must look at the listing file and determine what errors were committed. Then, the user should go back to the beginning of the cycle and correct the errors.

The next action, after successful completion of the hierarchy edit/compile cycle, is to enter the "mid" edit/compile cycle. The same course of actions will be followed here as were followed in the previous cycle. The one big, noticeable difference is that a partial mid description will already exist. This is so because the hierarchy compilation process was able to glean enough information from the hierarchy description to fill out parts of the mid description. One useful thing to know is that any information entered in the mid cycle will not be lost. Successive

recompilations will save the data entered in the first attempt. This service is provided to keep the user from having to enter the information more than one time. Obviously, if the information changes, the user must update the description.

After successfully leaving the mid cycle, the next cycle is the data, or release, cycle. In this part, the user must enter the descriptions of all the data that the mid compilation couldn't figure out. The purpose of this cycle is twofold, first it is supposed to compile the data description. Second, it is responsible for the translation of the MIDAS into the implementation language.

The translation will only take place if certain conditions have been met. The first is that all the three cycles have been successfully completed. The second is that an implementation language has been chosen. If the currently active level is not the main level, then the translation will take place. If the level is the main level then translation will only take place if all the separate levels have been taken care of first.

A sample project has been included in Sire to demonstrate its use. The project name is "Sample" and the levels are main, first and first_sub_one. A quick perusal of this project will be useful to the first time user.

D.7 Operation of DDS

TBD.

D.8 Operation of SRS

TBD.

D.9 Operation of Utility Tools

It is hard to fully describe all the Utility Tools that are built into Sire since, by nature, Utility Tools is very flexible and intended to change. It should be sufficient to

say that all the tools presently included in Sire are very, very simple in nature and should be easy to understand from just reading the title.

Appendix E

Installation and Maintenance of Sire

E.1 Introduction

Sire is fairly easy to install on a VAX/UNIX system that is running version 4.1 bsd. The rest of this appendix contains a list of the files needed for Sire and a brief description of those files. Also, instructions are given on how to move and install Sire.

E.2 File Descriptions

Currently all files listed below exist under /en/gcs83d/nettles/thesis/sire, on the AFIT SSC VAX.

bin	The location of all executable parts of Sire.
bin/data	The data compiler of the PDS tool.
bin/mid	The mid compiler of the PDS tool.
bin/utility_tools	The utility tools that can be used by every part of Sire.
bin/hierarchy	The hierarchy compiler of the PDS tool.
bin/pds	The PDS tool of Sire.
bin/sire	The Sire system driver.
exec	The directory that contains the source code to the Sire driver.
exec/Makefile	The make program that maintains the configuration of the directory.

exec/main.c The source code of the Sire driver.

pds The directory that contains the PDS tool source code.

pds/Makefile The PDS directory make program that maintains PDS configuration.

pds/data.y The YACC input file that contains all the source for the data compiler.

pds/data_table.c
The source for the data table code used in the data compiler.

pds/hash.c The hash source for the hashing functions used by hierarchy and mid compilers.

pds/hierarchy.y
The YACC input source used for the hierarchy compiler.

pds/main.c The main driver source of the PDS.

pds/mid.y The Yacc input source used for the mid compiler.

pds/midasall.l
The Lex input source used for the release compiler's lexical analyzer.

pds/midasdata.l
The Lex input source used for the data compiler's lexical analyzer.

pds/midashier.l
The Lex input source used for the hierarchy compiler's lexical analyzer.

pds/midasmid.l
The Lex input source used for the mid compiler's lexical analyzer.

pds/module_table.c
The module_table source used by the hierarchy compiler.

pds/release.y The YACC input source used for the release system that is triggered by the data compiler.

pds/tree.c The tree builder source used by the hierarchy compiler.

projects — The directory under which all the projects exist.

projects/.globals.h — The file which contains the global header file that is used by the translation module that generates C source code.

globals — The directory which contains things of global importance.

globals/Makefile — The global make file configuration maintainer.

globals/errors.h — The error definition header file.

globals/globals.h — The global definition header file.

globals/screen.c — The source file which contains both menu() and report_errors().

globals/struct.h — The structure and definition header file.

globals/utilities.c — The source file for many small utility programs of global use.

tools — The directory containing the utility tools.

tools/Makefile — The directory configuration maintenance file.

tools/utl_tools.c — The source code for the utility tools.

E.3 Maintaining Sire

Sire was developed in many small parts for the purpose of easy development and maintenance. Also, the design of Sire was run through the PDS system in order to produce documentation for the implemented parts of the design. This provides for a fairly good level of documentation. It is hard to describe the proper method for maintenance of Sire since the recommendations given earlier in the main body of the thesis call for the redesign and reimplementation of the

entire system. However, should some hardy soul wish to wade through the code and fix the system, the maintainer should become thoroughly familiar with the C language, YACC, and LEX for this is where the complexity of the system lies.

The majority of the source code is very simple and is broken into small modules. Also, there is seldom any occasion in which the calling level ever goes past three or four functions at a time. This means that the tracing of the system flow will be fairly easy.

E.4 Moving Sire

Moving Sire should prove to be a fairly easy exercise. All that need be done is to move the directory "sire", which currently exists under /en/gcs83d/nettles/thesis, to whatever final location is desired. Then, the definition of HOME in the globals/struct.h file must be changed to reflect the new location. Finally, the makefile in the sire directory must be invoked by giving the following command, "make sire >& serr &<CR>". This will remake the whole system in background mode and put any errors in the file "serr". There shouldn't be any errors.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S): AFIT/GCS/MA/83D-5	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION: School Of Engineering; 6b. OFFICE SYMBOL (If applicable): AFIT/ENC	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State and ZIP Code): Air Force Institute of Technology, Wright-Patterson AFB, Ohio 45433	7b. ADDRESS (City, State and ZIP Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION; 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State and ZIP Code)	10. SOURCE OF FUNDING NOS. — PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT NO.
11. TITLE (Include Security Classification): see box 19	

12. PERSONAL AUTHOR(S): David W. Nettles, B.S., 1lt, USAF

13a. TYPE OF REPORT: MS Thesis	13b. TIME COVERED: FROM _____ TO _____	14. DATE OF REPORT (Yr., Mo., Day): 1983 December	15. PAGE COUNT: 147

16. SUPPLEMENTARY NOTATION

Approved for public release: IAW AFR 190-17.
LYNN E. WOLAVER 7 Feb 84
Dean for Research and Professional Development
Air Force Institute of Technology (AIC)
Wright-Patterson AFB OH 45433

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB. GR.	
9	2		Software Development, Software Engineering, Automatic Program Generation

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Title: SIRE: AN AUTOMATIC SOFTWARE DEVELOPMENT ENVIRONMENT

Thesis Chairperson: Patricia K. Lawlis, Captain, USAF

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT: UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS ☐	21. ABSTRACT SECURITY CLASSIFICATION: UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL: Patricia K. Lawlis, Captain, USAF	22b. TELEPHONE NUMBER (Include Area Code): 513-255-3636 — 22c. OFFICE SYMBOL: AFIT/ENC

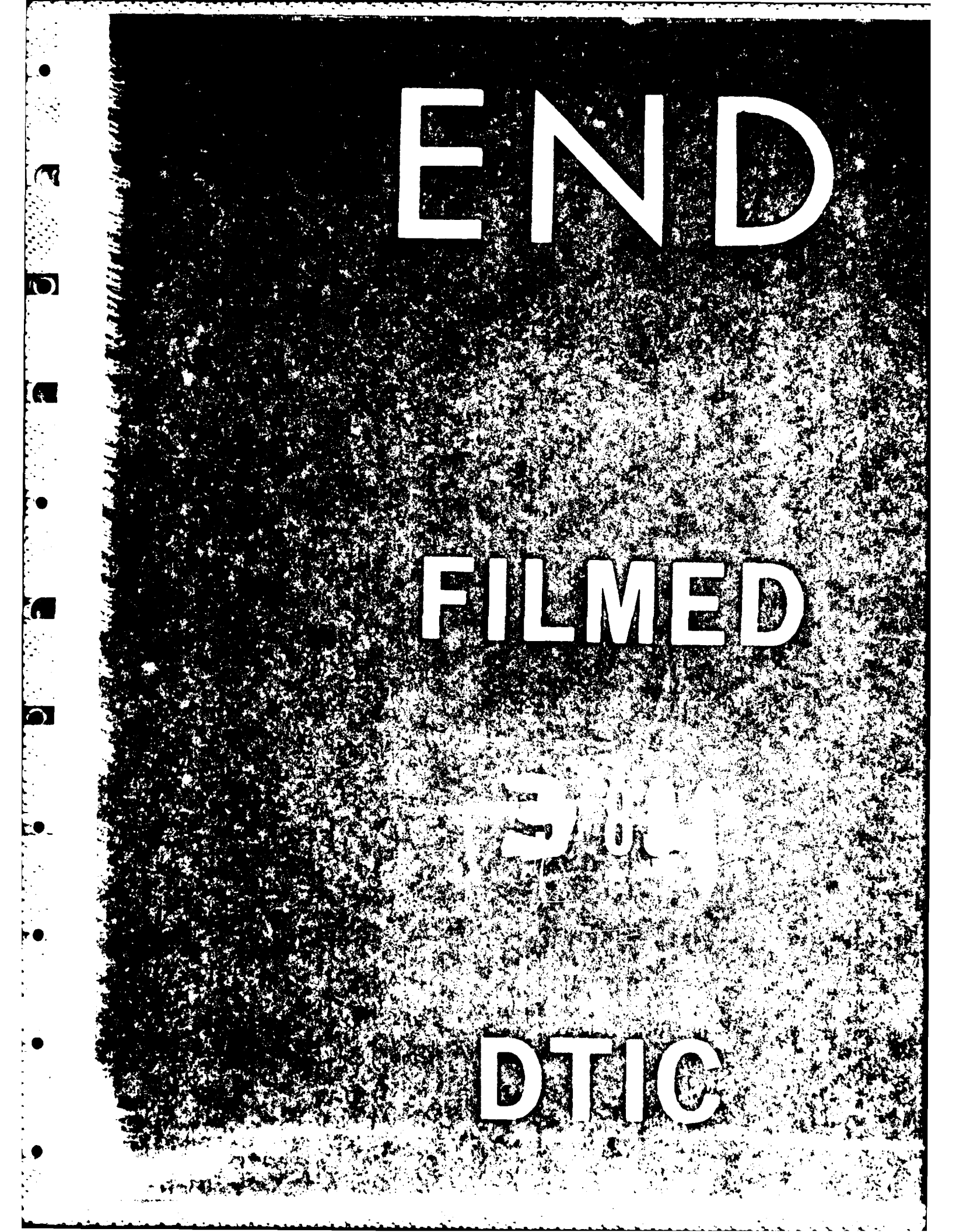
DD FORM 1473, 83 APR — EDITION OF 1 JAN 73 IS OBSOLETE.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

ABSTRACT:

The objective of this thesis is to perform the preliminary design and partial development of an automated software development environment (ASDE). This environment, called Sire, is intended to support the design and production of software using automated and interactive tools. Sire is to be a system that aids the software designers and programmers through the use of an integrated and flexible set of tools that are intended to reduce the amount of work that is done by humans. This reduced workload will free the system designers/implementors for more productive work.

As part of this investigation, a partial implementation of Sire is accomplished. This implementation allows the user to input a system design in a specification language. Sire will then produce a correct source program shell for the user to use for detailed design and implementation.


END
FILMED
DTIC